Microsoft® The Extensible Storage Engine

Exchange 5.5 (ESE97)
Windows 2000 Active Directory (ESENT)
Exchange 2000 (ESE98)

Draft 1.3

Stuart Padley
Microsoft Regional Escalation Centre
Northern Europe

Microsoft Confidential

© 1999 Microsoft Corporation. All rights reserved.

The information contained in this White Paper represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

BackOffice logo, Microsoft, Win32, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries..

Other product or company names mentioned herein may be the trademarks of their respective owners.

Microsoft Corporation • One Microsoft Way • Redmond, WA 98052-6399 • USA

Contents

MICROSOFT® THE EXTENSIBLE STORAGE ENGINE....1

EXCHANGE 5.5 (ESE97) WINDOWS 2000 ACTIVE DIRECTORY (ESENT) EXCHANGE 2000 (ESE98)....1

CONTENTS....3

RESEARCH / ACADEMIC / HOW TO....8

Draft revision history....9

ACKNOWLEDGMENTS....10

Exchange Product Group....10
Microsoft Product Support Services....10
Microsoft Consultancy Services....10
State of Document....10

INTRODUCTION....11

Concepts....11
History....12
Versions of ESE....12
What Changed in Exchange 2000....12
I/O has improved....12
Customer based improvements....12
Error detection during recovery....13
Log and Patch file checksumming....13
A faster eseutil....13
What's new in Exchange 2000....13
Multiple Databases....13
The Streaming file....13
An evolved backup and restore....14

DESIGN PHILOSOPHY....15

What if we crash?....15
Going to disk is expensive....15
Self tuning....15
Continuous operation....15

ESE DATABASE STRUCTURE....16

Inside the database....16
The tree of trees....16
B-trees....16
Pages....16
The checksum....17

-1018....18

Two categories of –1018....18
Transient fault....18
Hard fault....18
Retry Logic....19
What should I do if I get a transient –1018....19
What should I do if I get a hard –1018....20
A hard error can still be transient....20
Verify physical integrity whilst the database is offline....20
No physical damage to the data in the database....21
Physical damage to the data in the database....21

Recovery after a hard –1018, the result of physical damage....21

B-TREE....23

Balance....23
Keeping a B-tree balanced....23
Split....23
Merge....23
B+Tree....24
Fan out....25

INDEXES....26

Secondary index....26
Changing your view....26
Creating a secondary index....26

ESE DATABASES - LONG VALUES....27

The Long Value Root bug....27
Orphaned Long Values....27

RECORD FORMAT....29

Fixed Columns....29
Variable Columns....29
Tagged Columns....29

MULTIPLE DATABASES....30

Pros....30
Flexibility....30
Eseutil faster....30
Only those effected....30
Cons....30
More free space, more disk space....31
Single instance storage....31
Memory Usage....31
Complexity....31

STORAGE GROUPS....32

Unique Settings per Storage Group....32
Global Resources....33
Single set of logs....33
Information Store will always start....33

STREAMING FILE....34

EXIFS: THE EXCHANGE INSTALLABLE FILE SYSTEM....36

Inbound....37
Outbound....37
Space allocation in the streaming file....37
Recovery and reserved pages....38
Filesystem semantics....38

TRANSACTIONS....39

The Version Store....40
Rollback....40
Write-conflict detection....40
Repeatable reads....40
Deferred before-image logging....40
JET_errVersionStoreOutOfMemory....40
The version store and Multi-user considerations....40

LOGGING/RECOVERY.....43

We don't want.....43
We do want.....43
Theory.....43
First.....43
Second.....43
Third.....43
So, where is my mail?.....44
Consistent or Inconsistent.....44
Log files.....44
What is in a log file?.....46
Header.....46
Log Records.....46
Logfile Checksumming in Exchange 2000.....47
Two types of LRCK failure.....48
Torn write.....48
Data damage.....49
Manual log checksum verification.....50
Modifying the database.....50
Recovery.....51
The Checkpoint (EDB.CHK).....52

IMPLICATIONS OF PHYSICAL LOGGING.....53

Delete edb.chk.....53
Delete edb.log.....54
Write-back caching.....54

RECOVERY ERROR DETECTION IN EXCHANGE 2000.....56

Before and after timestamps.....56
Impossible to run soft recovery on restored database.....56
Log Required.....56
Database Mismatch.....57

BACKUP AND RESTORE IN EXCHANGE 5.5.....58

The patch file.....58
Backup the database.....58
Log File housekeeping.....58
Restore the database.....58
Hard vs. Soft recovery.....59

BACKUP AND RESTORE IN EXCHANGE 2000.....60

What changed in Exchange 2000.....60
Client Setup.....60
The Backup process.....60
Log Truncation.....61
Backup Granularity.....61
The Restore process.....62
Restore Granularity.....62
The 16th instance.....62
Administrator Actions.....63
HrESERestoreComplete.....63
Initialization.....63
Log File Preparation.....64
Log File Replay.....64
Finishing the Restore:.....64

SNAPSHOT BACKUPS.....66

The Snapshot backup application process.....66

Checking Physical integrity *67*
Pausing Database Writes *67*
Restore of a snapshot backup *67*

SPACE MANAGEMENT **68**

Space in the database file 68
Owned space *68*
Available space *68*
Why is my (4.0/5.0) database so big? 68
Online defragmentation 69
Offline vs. Online Defragmentation 69
Online Defragmentation in Exchange 2000 70
Free space discovery 71
ESEUTIL /MS 71

BUFFER MANAGEMENT **75**

Dynamic Buffer Allocation 75
Design goals *75*
General rules *75*
Exceptions *76*
Formula *76*
The LRU-K replacement algorithm 76
Page Victimization / Page Replacement Policy *76*
LRU *76*
LRU-K *76*

ESEUTIL **78**

How fast is Eseutil 78
ESEUTIL in Exchange 2000 78
Changes to ESEUTIL in Exchange 2000 78
No more /IS, /ISPRIV, /ISPUB, /DS switches *78*
Defrag (/D) does not shrink the streaming file *78*
No more /V /X switches with Integrity (/G) or Repair (/P) *78*
Verbose output goes to INTEG.RAW *78*
Integrity/Repair support streaming files *78*
Integrity is multithreaded *78*
Orphaned LVs are no longer ERRORs *79*
Do not run Platinum repair against an Exchange 5.5 database *79*
Eseutil /mh priv1.edb and Eseutil /mh priv1.stm 79
Database (.edb) file *79*
Streaming (.stm) file *80*
Eesutil /ml edb.log 82
Eseutil /mk edb.chk 83

ESEFILE **85**

Large and Fast File copy 85
Usage *85*
Checksumming an entire database 85
Usage *85*
Checksumming an individual page 85
Usage *85*
Output *85*
Large File deletion 85
Usage *85*

ESE PARAMETERS **86**

TO DO **88**

Bibliography and References 88

Misc – Notes to myself....88
Hardware support for ESE....90
SQLHDTST....90
What is Escrow....90
Exchange 2000 ESEUTIL retail options; Exchange 2000 ESEUTIL debug options....90
Exchange 2000 ESE Perfmon counters....90

Research / Academic / How to

THE PAGE HEADER....16
HOW ESE CALCULATES THE CHECKSUM....17
WHAT DO THE VALUES IN A –1018 ERROR MEAN....18
Layout....18
dbid....18
pgno....18
dbtime....18
EVENTS RETRY LOGIC CODE WILL LOG:....19
For each failed read....19
A Succesful retry....19
A hard –1018....19
USING ESEFILE TO VALIDATE CHECKSUM OF INDIVIDUAL PAGE....20
Usage:....21
Output:....21
THE HISTORY OF THE B-TREE....23
SQUEAKY LOBSTER....24
ZENO'S PARADOX....26
HOW DOES PROPERTY PROMOTION WORK?....34
Content received via MAPI....34
Content received via Internet Protocol....34
State 0 conversion....34
State 1 conversion....34
State 2 conversion....35
'Application/ms-tnef' content....35
Other special cases....35
ACID....39
Atomicity....39
Consistency....39
Isolation....39
Durability....40
SNAPSHOT ISOLATION....41
Repeatable reads....41
Serializable transactions....41
SCATTER/GATHER I/O....43
ELEVATOR ALGORITHM....43
EDBTMP.LOG....45
Circular logging on....45
Circular logging off....45
Then....45
Why?....45
THE LRCK....47
WHAT IS A TORN WRITE?....49

PROBLEMS WITH THE WAY ESE97 ALLOCATES SPACE.....71

What is a space tree?.....71
When are empty page extents added to the space tree?.....72
When is space allocated?.....72
How is space allocated?.....72
Locality of pages:.....72
The initial problem:.....72
Resolution to the initial problem:.....73
The second space tree problem:.....73
The resolution to the Exchange 5.5 Service Pack 2 space tree problem.....74
The Space Manager in Exchange 2000.....74

Draft revision history

Date	Ver.	Comments
22rd Nov 1999	1.0	First draft release.
22rd Nov 1999	1.1	Miscellaneous grammatical changes.
23rd Nov 1999	1.2.1	Over 60 small grammatical and technical changes. Changed 'History of the B-Tree section' to reflect the Russian got there first (Thanks to Luke Brennan and Stuart Kostalas
23rd Nov 1999	1.2.2	Added documentation of fields in eseutil /m? About 20 other grammatical changes spotted by juliet dimmick
06th Nov 1999	1.2.4	General grammar and readability improvements
06th Dec 1999	1.3	Integrated ESE event messages into document

Acknowledgments

Many thanks to the following people for their time and enthusiasm, no whitepaper is produced without many people going out of their way to assist.

Thank you to Adam Green for ‘absorbing’ me into the JET Team during the summer, and to Andrei Marinescu who not only kindly shared his office during this time but who was also always available to explain the inner working of JET whenever I needed help or guidance.

Thank you to Laurion Burchall and Phillip Hupf whose Microsoft Exchange Conference 1999 sessions provide the backbone for much of whitepaper.

Thank you to the members of the JET team (and others) who spend many hours answering the questions raised through JetTalk by those who implement and support this technology.

Exchange Product Group

Laurion Burchall
Andrew Goodsell
Adam Green
Phillip Hupf
Jonathan Liem
Cheen Liao
Andrei Marinescu

Microsoft Product Support Services

Luke Brennan
Stuart Kostalas

Microsoft Consultancy Services

Peter Nilsson

State of Document

This is the first released draft of this document; at this time all of its contents are strictly Microsoft Confidential.

If there is inaccuracies either grammatical or technical then please contact me so they may be corrected for future releases, additionally if you would like to see either more detail or just a section explained better then I would like to hear from you.

This current version of this document can always be found at \\emsweb\docs\docs\ese wp.

Stuart Padley
(stuartpa@microsoft.com)
22nd November 1999

Introduction

An Exchange Storage Engine
An Embedded Storage Engine
An Excellent Storage Engine
An Everywhere Storage Engine

An Extensible Storage Engine

- What does the B in B-tree stand for?
- What is the LRU-K algorithm?
- What is Scatter/Gather IO?
- What is SI?
- Who was Zeno?
- Who discovered ACID?

If you have ever wondered the answers to these questions, or may be you just want to get a good understanding of the database technology behind products such as Exchange Server and Active Directory then this white paper will give you a solid understanding of the database technology known as the Extensible Storage Engine.

Today ESE is used by the following services:

- Exchange Information Store
- Exchange Directory Service
- Exchange Security Key Management System
- Exchange Site Replication Service
- NT Directory Service
- NT Security Management
- NT WINS, DHCP
- Hierarchy Storage System

Concepts

This white paper will look at the 6 core concepts that make up ESE

- The Database Structure
- Transactions
- Transaction logging and recovery
- Backup and Restore
- Space management
- Buffer Management

We will take each of these concepts and have a deep look into how they have been implemented in ESE. Where necessary we will delve into the academic grounding for the design decisions that have taken in the development process of ESE.

History

A while back ESE used to be called JET (Joint Engine Technology), in fact its full internal code name was JET Blue. JET Blue is not the same as the JET that maybe found in Access (known as JET Red). The reason they share the same name is due to a plan a very long time ago to converge a number of Microsoft databases into one common technology platform, this never materialised and ESE has become its own technology that has evolved over the last four or so years.

ESE is a multi-user ISAM database with full DML (Data Manipulation Language) & DDL (Data Definition Language) capability. ESE allows applications to store records and create indexes to access those records in different ways. It is optimised for performance, optimised for very fast retrieval of data.

It is not a query engine, ESE does not directly support SQL, an application level layer such as OLEDB or the STORE process is needed to provide this type of access.

ESE is simply a very fast and highly optimised engine for storing loosely formatted semi structured data.

Versions of ESE

Today there are three versions of ESE currently in active development, each version has it's own code tree. In order of the development cycle they are

- ESE97 (ese.dll in system32 directory) – shipped in Exchange 5.5, although no new functionality will be added to ESE97 it is still supported by the Quick Fix Engineers.
- ESENT (esent.dll in system32 directory) – to be shipped in Windows 2000
- ESE98 (ese.dll in \exchsrvr\bin directory) – to be shipped in Exchange 2000. ESE98 sees a log file format and record format upgrade although fortunately these upgrades will be pretty much transparent to the user, i.e. no lengthy upgrade process at the ESE level as there in Exchange 5 to 5.5.

What Changed in Exchange 2000

I/O has improved

Time has been spent working to reduce the number of times ESE has to go to disk. A disk access is approximately 200,000 times slower than a comparative memory access.

I/O has been improved, in fact in unofficial tests running ESE98 (simply replacing ese.dll) on an Exchange 5.5. box sees a like for like performance improvement of about 30% when run through the internal standard test suites at Microsoft.

Customer based improvements

The rest of the major improvements have been done based on customer data in feedback from PSS. Human error in disaster recovery scenarios is without doubt one of the biggest call generators in PSS.

The side effects from mistakes in recovery can some times be obvious (i.e. an information store which fails to start) or can be invisible for sometime, sometimes weeks or months.

Error detection during recovery

Various forms of error detection for recovery mistakes have been introduced. See "Recovery Error detection in Exchange 2000" for the full detail.

Log and Patch file checksumming

Whilst we have always (or almost always) checksummed each page in the database as it is backed up stopping a backup in the case a checksum fails, this has not been done for the logs files.

Exchange 2000 introduces log record checksumming (internally known as the LRCK). The log record checksums are verified during backup and recovery.

Additionally we will also verify the pages in the patch file as they are copied to tape. So all data in an Exchange 2000 database will be verified for physical integrity as it is sent to tape.

A faster eseutil

The integrity check (eseutil /g) and repair (eseutil /p) has been made multi-threaded.

Making eseutil multi threaded increases the speed of these operations three or four fold.

As we have seen databases (.edb files) increase in size some now approaching the 200Gb mark these improvements are necessary.

What's new in Exchange 2000

Multiple Databases

Time has been spent removing perhaps the biggest block in practical terms to scaling a single Exchange Server box.

Exchange 5.5 allows the information store to have a single set of transaction log files and a single Private Information Store and/or a single Public Information Store.

Exchange 2000 sees those barriers broken down with the introduction of Multiple Databases and Storage Groups.

The Streaming file

Exchange 5.5 saw the information store use MAPI as it's primary storage format.

With the emergence of more and more clients using internet protocols Exchange 5.5 is spending a lot of time converting data from it's native internet format (i.e. MIME) to the rich text stored in the MAPI database.

In fact much of the time the data is never used in it's stored MAPI format and is converted on the way into the store and converted back again each time it is read.

Exchange 2000 sees the introduction of the streaming file, the native content store.

An evolved backup and restore

With all these new features Exchange 2000 has had to evolve backup and restore so it can support many attached databases, many storage groups and the streaming files.

Backup and Restore support in Exchange 2000 is greatly improved giving us great flexibility in what we want to achieve.

Design Philosophy

Before we go deep into ESE let us first outline the key principles behind the design of ESE. These principles we need to bear in mind as we go through the rest of this document.

What if we crash?

The most important design principal, everything we do has to be designed and written from the standpoint that the system might crash at any time.

ESE needs to be able to survive crashes; it must protect the data and recover so the data remains consistent and accessible.

Going to disk is expensive

ESE recognises that going to disk is devastatingly expensive on its performance. A visit to disk to retrieve 4k of data is about 200,000 times slower than getting the same data from memory.

ESE is willing to consume extra CPU cycles and will use almost all the memory in the machine to avoid having to go to disk.

Self tuning

The database engine should be self-tuning. Wherever possible it should regulate its use of log files/memory etc. with the bare minimum of Administrative interference.

Continuous operation

And of course ESE is designed for 24 hours, 7 days a week uninterrupted continuous operation.

ESE database structure

Inside the database

Within the huge big database file (the .edb file) everything stored is contained within B-trees. So from a database perspective a table in ESE is a collection of B-trees.

Each table comprises of one B-tree that contains the data, there may also be many secondary index B-trees. Index B-trees provide different views on the data.

Finally if a column/field in a table becomes too wide to store in the data B-tree it gets broken out into a separate B-tree called the long value tree.

The tree of trees

The definition of these tables and their associated B-trees is stored in another B-tree called the system catalogue. Loss of the system catalogue is pretty terminal, so much so ESE actually keeps 2 identical copies of this B-tree in your database, one starting at page 4 the other at page 24.

B-trees

B-trees are designed to provide fast access to data on disk.

As we know from the fundamental designed principles going to disk is expensive therefore, a B-tree is designed so ESE can get to the data it wants using the minimum number of Disk I/Os. How? A B-tree is broken into 4k pages.

NOTE: ESENT uses 8k pages

Pages

Each of these 4k pages contains either pointers to other pages or the actual data that is being stored in the B-tree. The pointer and data pages are intermixed within the file.

To increase performance wherever possible the pages are cached in a memory buffer for as long as possible (see the section “Buffer Management” for more detail on this) reducing the need to go to disk.

The page header

Each page starts with a 40-byte page header. The page header contains the following values:

Checksum	*- See next paragraph*
pgnoThis	*- The page number of the page*
DbtimeDirtied	*- The Dbtime the page was last modified*
pgnoPrev	*- The page number of the adjacent left page*
pgnoNext	*- The page number of the adjacent right page*
ObjidFDP	
cbFree	*- The number of bytes available on the page*
cbUncommittedFree	*- The number of uncommitted bytes available on the page*

ibMicFree	*- itagMicFree*
fFlags	*- Flags for internal use*

The checksum

When a page is ready to be written to disk the *last* thing ESE does is compute a checksum based on the data on the page and writes this checksum to the page header.

How ESE calculates the checksum

Below is the code ESE uses to calculate the checksum, as you can see it is very simple.

This code was posted to JetTalk by development to indicate the simplicity of the checksum; however the code must be treated as MSINTERNAL.

```
//  ================================================================
static ULONG UlChecksumESE( const BYTE * const pb, const ULONG cb )
//  ================================================================
        {
        const ULONG   * pul                    = (ULONG *)pb;
        const ULONG   * const pulMax           = pul + (cb/sizeof(ULONG));
        ULONGulChecksum = 0x89abcdef;

        // skip the first four bytes as it is the checksum
        goto Start;

        while ( pul < pulMax )
              {
              ulChecksum ^= pul[0];
Start:
              ulChecksum ^= pul[1];
              ulChecksum ^= pul[2];
              ulChecksum ^= pul[3];
              ulChecksum ^= pul[4];
              ulChecksum ^= pul[5];
              ulChecksum ^= pul[6];
              ulChecksum ^= pul[7];
              pul += 8;
              }
        return ulChecksum;
        }
```

Each time the page gets read off disk the *first* thing ESE does is re-compute the checksum based on the data just read off the disk and compares it to the checksum value saved in the page header.

If the calculated checksum matches the checksum on the page header, ESE then checks the page number of the page we got off the disk to ensure it is the page number we asked for; if either of these simple tests fail ESE returns a -1018 error.

-1018

These -1018 errors, or Jet Read Verify errors, what are they? Essentially ESE is double-checking every piece of data it gets off the disk.

When ESE returns a -1018 error it is saying that the data off the disk was not the data it sent to the disk, and it knows this because ESE has done its own check.

Two categories of –1018

In the Exchange 4.0/5.0 days when ESE returned a –1018 we understood that the data must be damaged on the disk.

Since then we have a better understanding of the sorts of things that have typically been causing -1018 errors and have found that they come in two categories.

Transient fault

A transient fault, caused possibly by heavy load that exposes some bug in the device driver software or controller firmware. When we look closely at the disk surface we see that the data is actually OK, but for some reason the read returned the wrong data.

Hard fault

A hard fault; here the data is actually wrong on the disk surface, here we assume something is actually broken in the way the data was written to the disk or the disk became damaged after ESE wrote the page to disk.

What do the values in a –1018 error mean

Layout

-1018 ((dbid:pgno dbid:pgno) (dbtimehigh-dbtimelow) (dbtimehigh-dbtimelow))

dbid

Database id: e.g. 1 for priv.edb and 2 for pub.edb.

pgno

Page number

dbtime

A per-page time indicator, showing when the page was modified. A dbtime is 8 bytes, so it is printed as two 4-byte values

The first dbid:pgno combination is for the page we were trying to read; i.e. the page we requested. The second dbid:pgno combination gives the pgno on the page we actually read off disk. Each page in the database has its page number written in the page header. We compare this page number with the page number we requested to make sure that we received the correct page. A mismatch in page numbers causes as -1018 error.

If the two dbid:pgno combinations are the same it means that the page number on the page we read was correct, but the checksum failed. We store a checksum of the page in the page header. When the page is read off disk we compute the checksum of the data

we read in and compare it with the checksum in the page header. If the two checksums don't match a -1018 error is returned.

Retry Logic

In fact it turns out that quite of a lot of -1018s fall in the first category of 'transient' faults. The data appears correct on the disk, but due to a problem in the disk sub-system that gets exposed, for example during a period when the controller is particularly stressed, the read operation returns the wrong data.

So in Exchange 5.5 Service Pack 2 we added what we call retry logic; if a read fails with a -1018 error, ESE then retries the read up to 16 times, with a short pause between each read.

ESE does this in the hope that a transient problem might go away, so either you get a successful read within the retry window, or after 17 failures we assume that in fact we are looking at a hard error.

Whatever happens is recorded in the application event log.

Events retry logic code will log:

For each failed read

```
Synchronous read page checksum error -1018 ((1:31358 1:31358) (0-
257205199) (0-257246802)) occurred. Please restore the databases from a
previous backup.
```

A Succesful retry

```
A read of the database file M:\exchsrvr\MDBDATA\PRIV.EDB between offsets
0x0000000007A7F000 and 0x0000000007A7FFFF succeeded after 4 failed read
attempts. The error seen during the failed attempts was -1018. There is a
transient software or hardware problem affecting the database drive that
must be corrected to preserve database integrity. Contact Microsoft
Product Support Services.
```

A hard –1018

```
M:\exchsrvr\MDBDATA\PRIV.EDB between offsets 0x0000000007A7F000 and
0x0000000007A7FFFF failed after 16 failed read attempts with error -1018.
Please restore the databases from a previous backup.
```

If you see retries but you are not getting hard errors the databases are OK, they are not damaged on disk but you do need to investigate the cause of the problems in your disk subsystem.

What should I do if I get a transient –1018

If in the application event log you notice the retry code is being activated as it tries to get the correct physical data off disk and it succeeds then you can guarantee that there is not any physical damage to the data being read off disk, however you can guarantee that the disk subsystem is undergoing some problem.

This problem should be looked into with your hardware engineers. The events in the event log are telling you that the disk subsystem is not always returning to the operating system the same data that is on the disk. This can be an early sign of more serious

hardware failure in the future so the disk controllers, memory cache (both on the controller and the disk itself), the disk units and the connecting cables should all be investigated for their integrity especially when under high load.

What should I do if I get a hard –1018

You will notice if you get a hard –1018 because the process (i.e. store.exe for the information store) that is using the database that causes the error will be terminated by ESE.

In Exchange 5.5 Service Pack 2 or later in the application event log there will 16 events logged indicating the 16 attempts to read this page, the event indicates the reason why the read was unsuccessful, i.e. either the page number received was not the same as the page number requested or the checksums did not match.

After the 16th retry ESE will log an event indicating it is going to stop the process.

ESE is of the opinion that it is safer to stop and not let data known to be damaged into the database system. A B-tree structure in an ESE database is a complex structure and it could potentially be disastrous if bad data got into the database potentially affecting the consistency of the data in the entire database.

A hard error can still be transient

The first step that should be taken after a process has been stopped due to an –1018 error is to verify whether the data on the page is indeed damaged.

Let me explain, even though ESE will attempt to read a page 16 times in the hope it might get the correct data, it is entirely possible that the disk subsystem will simply return the page 16 times from the controller or disk read cache.

So although ESE asked for the page to be read 16 times, the disk subsystem might have actually only read the data from the physical disk once. It is possible that the one physical read was the operation that did not happen successfully causing the wrong data to be returned to ESE each of the 16 times.

Verify physical integrity whilst the database is offline

To verify whether the data on the page is physically intact whilst the database is offline enables the disk subsystem to be given a chance to return the correct page whilst not under any pressure.

To quickly validate the checksum for an individual page in the database you can use the utility Esefile. Esefile first shipped with Exchange 5.5 Service Pack 3, it is also available from Product Support Services.

Esefile contains no ESE code therefore it does even map ese.dll into its process address space, it does not initialise the ESE database and has no ability to make any modification to your database. Esefile will run at a rate (depending on your hardware) up to 80Gb/hr

Using ESEFILE to validate checksum of individual page

To validate the checksum of an individual page in an offline database you can use ESEFILE (first shipped in Exchange 5.5 Service Pack 3).

At the command prompt use the syntax below where <database> is the full path and file name of the database reporting the error and the <pageno> is the page number being reporting in the –1018 error.

The last line of the ESEFILE /D output will report whether the data is CORRECT (good on disk) or INCORRECT (damaged on disk)

Usage:

```
ESEFILE /D <database> <pageno>
```

Output:

```
…
00000000  2a c3 aa 89 01 00 00 00 01 00 00 00 00 00 00 00 *...............

00000010  00 00 00 00 00 00 00 00 01 00 00 00 c4 0f 00 00 ................

…
checksum: 0x5477CEA7 (1417137831) (INCORRECT)

or

checksum: 0x5477CEA7 (1417137831) (CORRECT)
```

After using Esefile to validate the physical integrity of the database you will know either:

- There is no physical damage to the data in the database
- There is physical damage to the data in the database

No physical damage to the data in the database

Follow the guidelines ‘What should I do if I get a transient –1018’.

Physical damage to the data in the database

If Esefile confirms that the checksum in a page header does not match the calculated checksum based on the data for the page then something has happened to that data outside ESE, the physical image of the data has become damaged either:

- On its way to the disk surface after ESE handed it over to the disk system.
- Whilst it has been on the disk surface

Before proceeding to recover ESE from this situation it is essential that the cause of the data damage is found and the necessary hardware is replaced.

Investigating the problem with the disk subsystem will probably be done by a hardware engineer, the 4 elements to look closely at are the disk itself, the disk controller, write-back memory cache and any interconnecting cables.

Recovery after a hard –1018, the result of physical damage

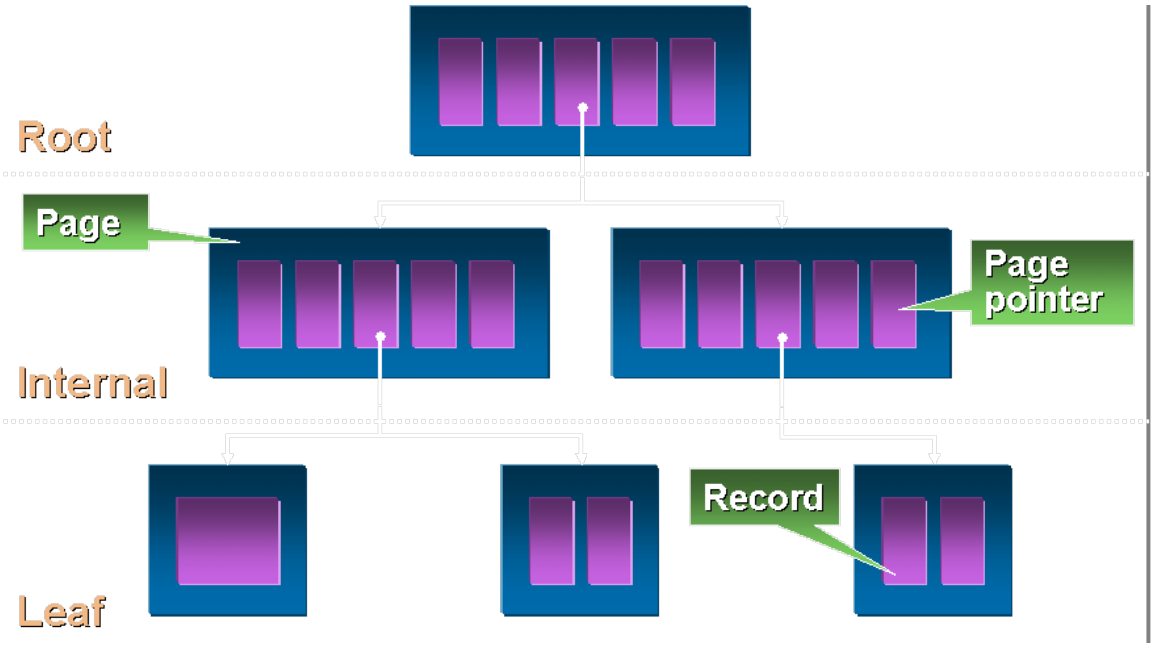
Once the problem with the hardware has been resolved it is time to recover the ESE database, For those databases that are running with ‘circular logging’ turned off there is only one option that can guarantee a no data loss solution, this is to restore from the last full/copy backup and allow the transaction log files to be rolled forward to bring the database to state it was at immediately before the crash.

If the database was running with circular logging turned on the log files will not be available to roll forward the transactions from the last backup, meaning if a restore was

performed all data since the last full/copy backup will be lost, if this is not an acceptable way forward then other methods of resolution may be adopted such as:

- Use the eseutil repair option (/p)
- Perform a ‘Move Mailbox’ for every mailbox on the server
- Use a utility such as ExMerge to move the data down to .pst files

B-tree



Balance

The B in B-tree is said to stand for Balanced, as you can see from this diagram the tree is Balanced, there is always the same number of nodes from the root of the tree to the leaf nodes at the bottom of the tree.

Keeping a B-tree balanced

It is the job of the database management system to keep the tree balanced at all times. The process of keeping a tree balanced is done when pages in the tree are either split or merged.

Split

When a page becomes nearly full roughly half the data on the nearly full page is placed onto a secondary page and an extra key is placed in the secondary page's parent (pointer) page. This is easy unless the parent page is also full. In this case we need to split the parent page and add a pointer to this page's parent page. Ultimately every pointer page all the way up to the root block may need to be split. If the root block needs to be split the tree must grow in height.

Merge

Conversely when a page is too empty it is merged with an adjacent page, the pointers in the parent page are updated and if necessary this page would also be merged. Ultimately if every pointer page all the way up to the root block is merged the tree will shrink in height.

To get to the leaf (data) node we start at the root node and follow the page pointers through until we get to the data we want that is stored in one of the leaf nodes.

The history of the B-tree

There are many data structures used in software engineering, simple ones such as the array, good for fast random access but slow for insertions and deletions. The linked list good for fast insertion and deletion but slow for sequential searches.

These simple structures have evolved into tree structures such as binary trees, red-black trees and splay trees all designed to provide 'near array speed' fast random access whilst allowing for 'near linked list speed' insertions and deletions

Two Russian mathematicians Adelson-Velskii, G. M., and E. M. Landis in 1962 published a paper on AVL trees, this is the founding paper in the field of storing data in balanced trees.

Their AVL tree (height balanced tree) is an improvement on the binary tree in that it is simply a binary tree that maintains the search condition while continually balancing itself. The effect of always balancing itself means that the AVL tree can always maintain a speedy search time, Knuth goes onto describe this work (in his book 'The Art of Computer Programming') as a "very pretty solution to the problem of maintaining a good search tree"

However all of the data structures described become inefficient when the data they contain is too large to remain in memory.

*The 'B-tree' structure and its associated algorithms where first published in a paper by Bayer and McCreight in 1972 ('*Transaction Processing: Concepts and Techniques'*: Morgan Kaufmann Publishers). The B-tree has become the storage structure used in almost all modern database implementations.*

The B-tree structure is an ideal data structure for large amounts of data that is inserted, updated and removed regularly and needs to be accessed quickly.

The B-tree structure makes the best use of two principles:

Accessing data on a disk is very slow compared to the time required to access data in memory, therefore making any I/O devastatingly expensive to the performance of an algorithm.

Read/write operations are best performed in fairly large chunks of data. Indeed, the more we read at a time, the better we'll do (within limits). Using blocks of 4 or 8k is understood to be optimal

B^+Tree

Although the trees inside an ESE database are commonly referred to as B-trees, they are in fact B^+trees.

A B^+Tree has all the characteristics of an ordinary B-tree, but additionally each data page in the B^+Tree has page pointers to its previous and next adjacent page. Although there is an overhead during insertion/split and merge operations to keep these pointers up to date, the pointers allow for faster sequential seeking through the data in the B^+Tree structure.

Squeaky Lobster

To see what ESE is really doing the development team have added a set of Microsoft Internal performance monitor counters to the Database object that give us in-depth performance data.

In Exchange 5.5 these extended counters are only visible when the following registry DWORD entry is set to 0x1:

HKLM\System\CurrentControlSet\Services\ESE97\Performance\Squeaky Lobster

In Exchange 2000 Beta 3 these counters are visible by default, there is no need for the above registry entry.

The counters' B+Tree Move Nexts/sec' and 'B+Tree Move Prevs/sec' show how often ESE is searching either left or right through the data pages at the bottom of the B+Trees.

The counter B+Tree Seeks/sec shows how often ESE is seeking down or up through the B+Tree pages to get from the B+Tree root page to the data page.

Fan out

A tree structure of a B-tree has very high fan out, ESE will store over 200 page pointers per 4k page, this allows us to have very shallow trees. This diagram above is a tree of height three; a tree of height four can store many gigabytes of data.

The high fan out means we can guarantee to reach any piece of data in a 50Gb table with no more that 4 disk reads (3 pointer pages and the data page itself).

In practice we get even better than this since there are only a few large tables in an Exchange Database, and for these tables the root page and at least most of the second level pages will be cached in memory.

Indexes

So a traditional B-tree is indexed in one particular way, using one key and the data must be retrieved that way.

For example, the records in the messages table are indexed on a message's unique identifier MTS-ID, however a user would probably want to view the data in the messages table ordered on a more human friendly value.

Secondary index

When we want to retrieve the data in different ways we use indexes. A secondary index is another B-tree, and it simply maps the chosen secondary key onto the primary key, so index B-trees are small compared to the data they index.

To understand how a secondary index gets used consider what happens when you change the way your messages are presented to you in folder.

Changing your view

If you change your folder view in Outlook so that subject instead of received time orders the view, Outlook through MAPI causes the store and ESE to build a new index (a secondary index) on your message folder table.

When you change views on a large folder for the first time, you will experience some delay, and if you looked closely at the server you will see a little spike in disk activity. But subsequently when you switch to that view again the index is already built and the response will be much quicker.

These secondary index B-trees live for 7 days (if they are not used) and then the store ages them out as a background operation.

Creating a secondary index

ESE allows an index to be created without actually having to take the table that is being indexed offline. To do this ESE uses the version store to keep track of all modifications made to the table whilst the index is being created and only at the very end of the index creation updates to the table are temporarily blocked to avoid Zeno's Paradox.

> ***Zeno's Paradox***
>
> *Zeno of Elea was the first great doubter in mathematics. His paradoxes stumped mathematicians for millennia and provided enough aggravation to lead to numerous discoveries in the attempt to solve them. One of the Zeno's most famous paradoxes was Achilles*
>
> ***The* Achilles**
>
> *The running Achilles can never catch a crawling tortoise ahead of him because he must first reach where the tortoise started. However, when he reaches there, the tortoise has moved ahead, and Achilles must now run to the new position, which by the time he reaches the tortoise has moved ahead, etc. Hence the tortoise will always be ahead.*

The fact that ESE allows these secondary indexes to be dynamically created and deleted is actually a major advantage for Exchange and one of the particular benefits in using ESE as the database subsystem for a messaging database.

ESE Databases - Long Values

A column or a record in ESE cannot span pages in the data B-tree. However there will be values (such as PR_BODY, the message body of a message) that will break the 4k boundary of a page, so for these values we need a way of storing the data; for this purpose we use a table's Long Value B-tree

If a piece of data comes into an ESE table that is too big to go into the data B-tree, it is broken up into chunks of almost 4K and stored in the table's separate long-value B-tree. The 'record' in the data B-tree contains a pointer to the long value. This pointer is named the long value ID (LID), e.g. the record just points to LID 256.

The Long Value Root bug

Each long value has a long value root, the long value root records the length of the long value.

There was a bug in the RTM (Release To Manufacturing) version of Exchange 5.5 where in rare circumstances during a B-tree split operation the LV root could get lost, this would lead to a logical corruption inside the database. (This bug has also been referred to as the 'Corrupt LV bug')

The 'long value root bug' was fixed by a hotfix, this was included in 5.5 Service Pack 1, and in fact long values corrupted in this way could be detected and repaired by the version of ESEUTIL (and ese.dll) that shipped in Service Pack 1 with no data loss. To detect whether a database is suffering from the 'long value root bug' run the Integrity check (eseutil /g /v /x > output.txt). If 'output.txt' contains entries similar to:

```
ERROR: corrupted LV(number) (lid changed without root)
```

Running repair (eseutil /p) will inspect each long value recalculating its length and fixing up its root without incurring any data loss

Orphaned Long Values

In order to save space we provide the ability for multiple records to share the same long-value. To do this a reference count is attached to each long-value. When the reference count reaches zero the long-value will be deleted.

However if we crash after de-referencing a long-value, but before expunging it from the database, the long-value will become "orphaned". Its reference count will be zero, but it will never be removed.

Orphaned long-values are invisible to anyone using the database because they are logically deleted, but still take up space because they have not been physically removed.

The integrity/repair functionality in ESEUTIL will detect orphaned Long Values but doesn't see them as corruption. If you use the /X option you will see output like:

```
ERROR: orphaned LV (lid 62322, size 80682). Non-corrupting error
ERROR: node [514410:0]: leaf node check failed
```

This example shows us the Long Value with LID 62322 has a reference count of 0, but has not been removed from the database. It is taking up 80,682 bytes of space in the database. The next line of output gives the page and location on the page where this particular LV starts.

The only way to remove orphaned Long Values in Exchange 5.5 is to do an offline defrag (ESEUTIL /D). In Exchange 2000, Online Defragmentation will remove orphaned Long Values.

Remember to look at the total space occupied by orphaned Long Values before suggesting an offline defrag. 1MB of orphaned Long Values in a 10GB database represents a negligible percentage of the total database space you will not gain much with an offline defrag.

Record Format

Tables in ESE contain records, a record is a group of columns. The maximum size of a record is approximately 4050 bytes (4096 bytes minus the size of the page header).

Columns fall into two categories:

- Columns that are expected to be present exactly once in each record with the possibility of a few NULL values, ESE calls these fixed and variable columns, these columns are represented in each record.
- Columns that either occur rarely or which have multiple occurrences in a single record, ESE calls these tagged columns, an undefined tagged column incurs no space overhead, similarly a tagged column may have repeated occurrences within the same record. If a tagged column is represented in a secondary index each distinct occurrence of the column is referenced by the index.

A table in ESE may contain:

- 126 fixed columns (column id 1 – 127)
- 127 variable columns (column id 128 – 256)
- 65,279 tagged columns (column id 257 - 64k)

Each column is defined as a 16 byte FIELD structure plus the size of the column name, An array of FIELD structures define the table, the column ID is the index value into the array. Search by column name requires a linear scan through the array if FIELD structures.

Fixed Columns

Contain a fixed length of data, each record occupies the defined amount of record space even if no value is defined.

Variable Columns

A variable column may contain up to 256 bytes of data, an offset array is stored in the record whose size is determined by the highest variable column set. Each column requires 2 bytes.

Tagged Columns

A tagged columns may contain an unlimited variable length of data, the column id and length is stored with the data. There is multiple value and sparse column support.

TODO: This is weak, needs better explanation

Multiple Databases

Exchange 5.5 allows each ESE database to contain 2^{32} (4,294,967,296) pages, each page is 4K (about 46 bytes is header information and 4050 bytes of data/index keys), this gives us an upper limit of 16TB per database which for all practical purposes is an 'unlimited' size.

Along with this potentially 'unlimited' size there is the downside that any manipulation (such as a integrity check) of these potentially large monolithic files would also take 'unlimited' time.

Exchange 2000 allows us to break down these potentially huge files into smaller entities allowing them to be managed and manipulated far easier.

Allowing the administrator to split up the size of the database into smaller entities has both good and bad points.

Pros

Flexibility

There is greater flexibility in scheduling backups. Backup and restore of multiple databases (although in different storage groups) can happen simultaneously. The ability to perform parallel backups could lead to a huge increase in backup and most importantly restore throughput when compared with Exchange 5.5.

Eseutil faster

Offline utilities will run much faster. Those (albeit few) customers who have circumvented any limitations imposed in backup and restore times in Exchange 5.5 by performing a 'snap' backup (more on that later) are still faced with the dilemma of maybe having to perform an operation using an offline utility, incurring downtime proportional to the size of the database.

Offline defrag (eseutil /d) runs at about 5Gb an hour, the maximum period of continuous downtime a company is likely to accept is usually 36 hours (a weekend leaving a few hours spare), therefore a database cannot grow any larger than about 180 - 200Gb (36 * 5).

Being able to split the database into smaller size removes this last limitation allowing a single server to potentially host terabytes of data.

Only those effected

Experience in PSS has seen that the largest cause of unscheduled downtime is media failure, and this is usually a disk drive(s) and less often a disk controller. Because we can split and place the databases on different logical volumes the only people affected by a media failure are those who database is hosted on the failed media.

Cons

Whilst there are many good points for multiple databases there are some things to watch out for:

More free space, more disk space

Every database contains an element of free space, the amount of free space varies depending on the ratio of insertions over deletions, free space in one database

cannot be used up by another, this will lead to a (although usually small) increase in disk space usage.

Single instance storage

One of the great features of ESE is single instance storage; this feature is put to good use in Exchange as messages are typically sent to more than one person.

Single instance storage means that the 'long values' (typically the message body and attachments) for any message will only be stored physically once in any one database no matter how many how many people that message has been sent to.

Single instance storage will be less and less effective as the users are spread over more databases.

Memory Usage

There will be a memory overhead for each database added to the Exchange Server. During initial testing we have seen about a 10Mb footprint required for each database. Therefore a server that wants to run with the maximum 90 databases will need just under 1Gb of physical memory just to start the Information Store.

Complexity

Perhaps the area that should be given most thought when deciding to increase the number of databases on a server is the increase in the Administrative burden.

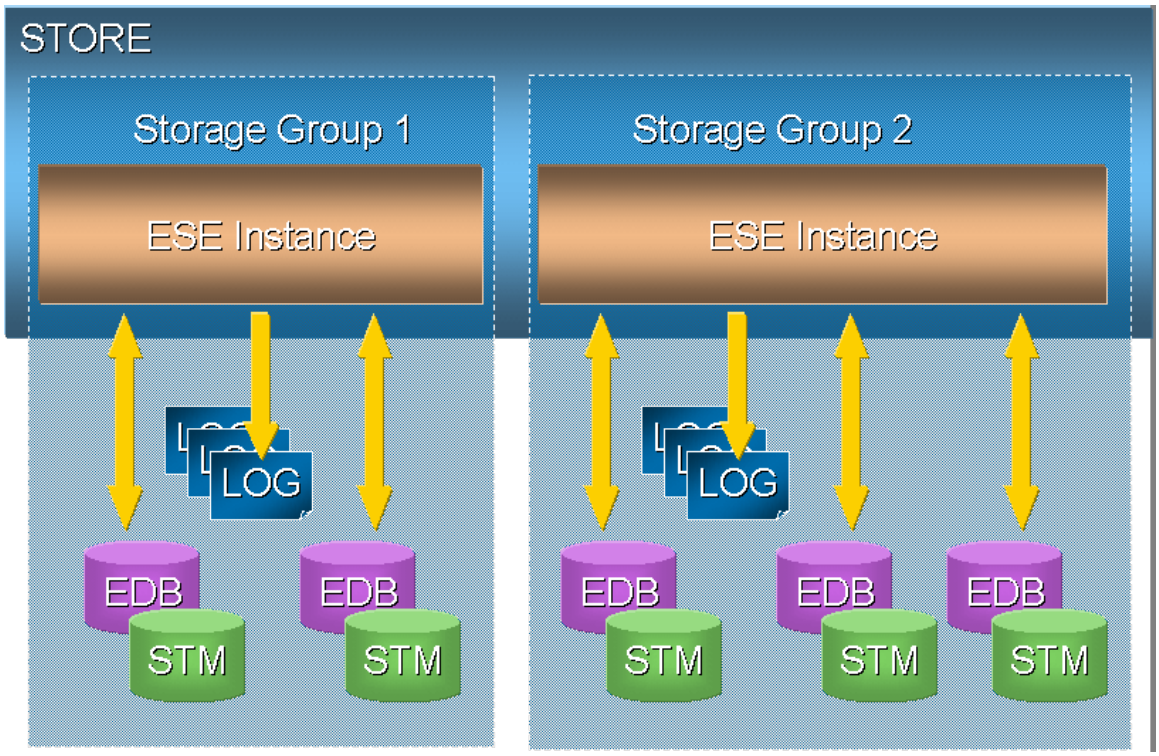
Having multiple databases and potentially multiple sets of transaction logs (each with that most important file called 'edb.log') will make things more complex.

Additionally any change to the logical location of a database will invalidate the current backups for that storage group.

Storage Groups

A storage group is a set of databases, the accompanying transaction logs and a few auxiliary files (i.e. internal temporary databases, the checkpoint file and reserve logs)

A single Storage Group will support up to 6 databases, and a single Exchange server may have up to 15 storage groups. Yes a single Exchange 2000 server may host up to 90 (15 * 6) databases.



From the diagram above we can see all the storage groups (they may be upto 15 of them) are hosted from the same store.exe process. Each storage group is represented by an ESE instance.

An ESE instance is largely autonomous and has it's own set of system parameters,

Unique Settings per Storage Group

The following are unique to each Storage Group:

- Circular logging (off by default).
- Temporary Databases
- Log files
- Check point file
- System paths
- Maximum Open Tables and Sessions

Global Resources

However the following resources are global to the entire process ESE is being hosted in:

- Cache size (Buffer manager)
- Version Store

So it is worth noting a hung transaction that causes all the memory in the version store to be used up will bring the whole store, all storage groups and databases to a halt.

Single set of logs

A storage group has a single set of transaction log files; all databases in that storage group share the log file set, each set of transaction logs will have an edb.log (Yes, there will be up to 15 files called edb.log on a single server).

Information Store will always start

At Information Store start-up each storage group is initialised and databases are recovered where necessary, any problems with a storage group will not affect the others from starting.

Indeed it is completely possible to have the information store start up with no storage groups running.

Streaming File

ESE has introduced in Exchange 2000 the facility for it's clients (i.e. the Information Store) to store raw data in it’s native content.

The repository for this raw data is known as the Streaming File. The streaming file has no B-tree overhead. The file contains 2 4k pages of header information and then raw data in 4k pages. This flat data structure is designed for Binary Large OBjects (BLOBs) of data that are unlikely to need content conversion and can be received and transmitted very quickly.

The traditional .edb file and its accompanying .STM file are a single unit, one is of little use without the other. So it is important to understand that a single database in the Exchange 2000 information store now contains two files, the .edb and the .stm.

All space usage (extents of 4k page numbers) and checksum data is stored in the .edb file.

A record (a record may represent a mail message) in the .edb file will contain a column that references a list of pages in the streaming file.

The information store supports the ‘property promotion’ of data held in the .stm (streaming file) to the .edb file, ‘property promotion’ enables folder views and indexes to be maintained efficiently.

For example a message streamed into the .stm file will have its properties such as sender, subject, date sent and received ‘promoted’ to the 'record' representing the message in the .edb file.

How does property promotion work?

Although not a function performed by ESE property promotion will determine where data is stored in an ESE database and is therefore an important concept to understand:

Content received via MAPI

If a MAPI client submits a message to the Store the contents of that message are stored in the EDB. If a non-MAPI client asks for this message the Store will do an on the fly conversion of the MAPI content to Internet format. (technically the Store does some of the conversion then calls IMAIL and IMAIL calls RTFHTML). None of this conversion is persisted - meaning that we don't move the data out of the EDB into the STM.

Content received via Internet Protocol

State 0 conversion

If an Internet Client submits a message to the Store the contents of that message is stored in the STM. Certain headers from the Internet message are duplicated into the EDB so the Store can find the message - this is state 0 conversion.

State 1 conversion

If a client asks for a property like PR_Subject or one of its many aliases then the store will "promote" all of the Internet message's header information into Properties - this is state 1 conversion.

State 2 conversion

If a client asks for attachment information then we go into state 2 conversion and the store creates a near duplicate (in MAPI form) of the Internet message. At this point the message is still in the STM but a lot of data needed for MAPI access is in the EDB as well. To reduce message bloat we have introduced the concept of "virtual" properties. A virtual property functions as a pointer to the actual data. A good example is attachment data. If the message came in through an Internet protocol the MAPI Attachment table will contain pointers to the offsets in the STM for this data.

If a client alters the message in a way that changes the MIME then we throw away the STM version of the message and keep the one in the EDB.

'Application/ms-tnef' content

However, if a Server receives Internet content that includes Application/ms-tnef (or winmail.dat if sent via Uuencode): The message initially goes into the STM but is then decoded and moved to the EDB. TNEF is an encoding format for MAPI messages on transports that do not support MAPI so the general principal that MAPI messages live in the EDB and Internet messages live in the STM still holds. The current functionality has the TNEF decoded before any of the MAPI properties are read.

Other special cases

There is also special case behaviour for S/MIME, DSNs, and MDNs because these Internet message formats interact with Outlook. The general rule holds but for each type there is something different the Store has to accommodate.

ExIFS: The Exchange Installable File System

IIS is the front-end service for Internet Protocols in Platinum, a set of protocol plug-ins in IIS talk to Exchange Server for incoming and outgoing mail.

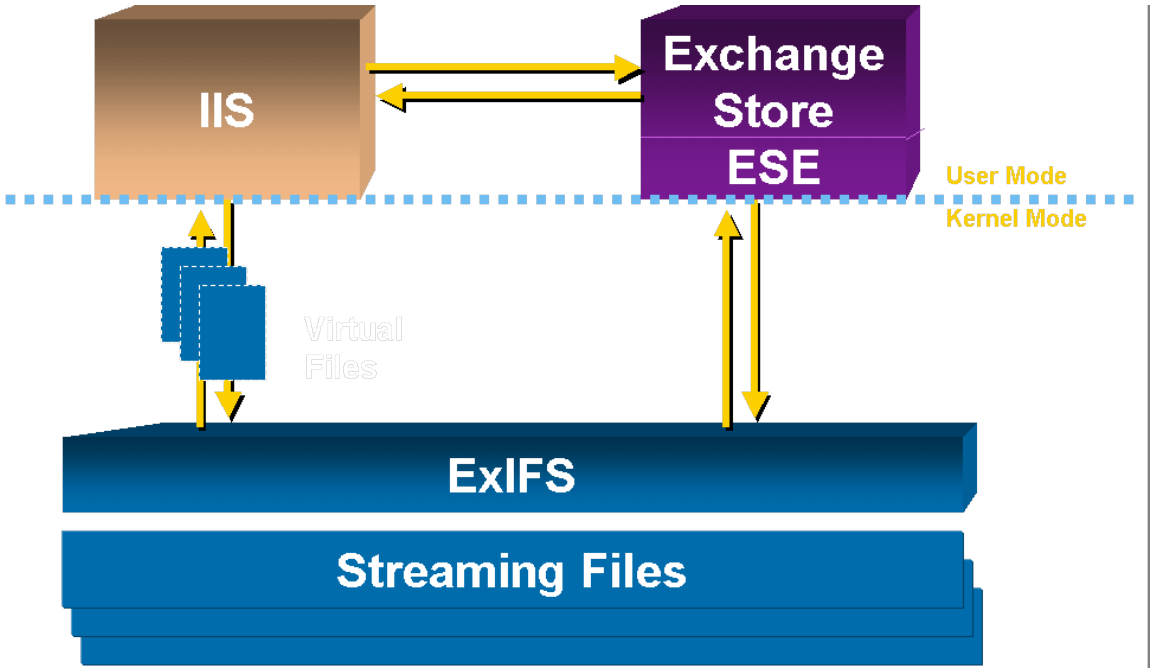
As we move away from proprietary protocols towards the open Internet protocols there is going to be a need to pass huge volumes of data between IIS and Exchange (and therefore the ESE database) very quickly.

In Exchange 5.5 this type of interaction has typically been done using NTFS files as an intermediary step (for example the IMCDATA\IN files to MTS-IN folders).

But as we move to Exchange 2000 we want to go faster... much faster. To do this we came up with the Exchange Installable File System, or ExIFS for short. ExIFS makes the stream file appear to IIS as lots of smaller files (called virtual files).

IIS obtains a 'handle' (a kernel object) to a virtual file and writes incoming data directly into the stream file through IFS, and vice versa outgoing messages are read directly from the stream file through IFS.

A message then becomes a list of pages (the numbers of which are held in the .edb file) in the streaming file and any properties needed for folder views etc. are promoted from the .stm file to the .edb file.



Here we see IIS interacting with the .stm file through ExIFS that makes the stream file appears as smaller virtual files.

On the right-hand side we see data moving up to ESE such as the checksum data and the promotion of properties from IFS to the Exchange Store.

IIS and the Store also communicate exchanging information such as the page numbers IFS placed a message on so they may be stored on the record representing the message in the store.

ExIFS (exifs.sys) is a kernel mode file system driver (so yes, at last Exchange can truly blue screen the machine)

Inbound

IIS creates a new file in IFS and writes to it placing the data on 'reserved' pages (see space allocation in the streaming file' below for description of reserved pages). IFS then returns the list of pages the message was written to.

The list of pages is passed to the Exchange Server process where they are stored in a record representing the message. ESE then commits the data on the reserved pages by logging the information to the storage groups transaction log files, at this point the pages have change from a 'reserved' state to a 'committed' state.

Page checksums for the data written to the stream file are calculated and stored in the .edb file.

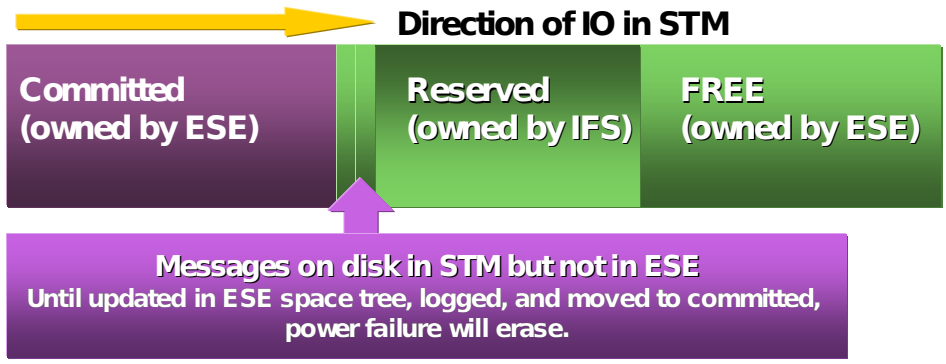
Outbound

The Exchange Server process gets the list of pages referenced by the message. This list of pages is passed to IIS; IIS opens a file in IFS using the list of pages.

The message is streamed out of the store quickly without conversion. Page checksums are not verified during transmission.

Space allocation in the streaming file

ESE knows which of the 4k pages in the streaming file are currently committed, ESE also know which pages are free and will allocate chunks of pages in 2mb contiguous blocks to IFS when it requests space, this space is known as 'reserved' space. From it's reserved space IFS allocate pages for new messages, if more space is necessary IFS will request further space from ESE.


Direction of IO in STM
Committed
(owned by ESE)
Reserved
(owned by IFS)
FREE
(owned by ESE)
Messages on disk in STM but not in ESE
Until updated in ESE space tree, logged, and moved to committed,
power failure will erase.

Therefore a page in the steaming file can be in one of the following states:

- Free - The space is free for use. Valid transitions are to 'reserved' state (for IFS) or directly to 'committed' state.
- Reserved - The space has been allocated for IFS to use. Valid transitions are to the 'committed' state or back to the 'free' state on a rollback or recovery.

- Committed - The space is consumed and unavailable for use. Valid transitions are to the ‘deleted’ state or directly back to the ‘free’ state on a rollback.
- Deleted – ‘committed’ space is being released. Valid transitions are back to the ‘committed’ state if the deletion rolls back or to the ‘free’ state if the deletion commits or after recovery.

Recovery and reserved pages

Because data on reserved pages in the streaming file has not yet been recorded in the transaction logs the data is not durable. Therefore should we crash, during recovery any pages that are marked as reserved in the streaming file are returned to ESE as ‘free’ pages.

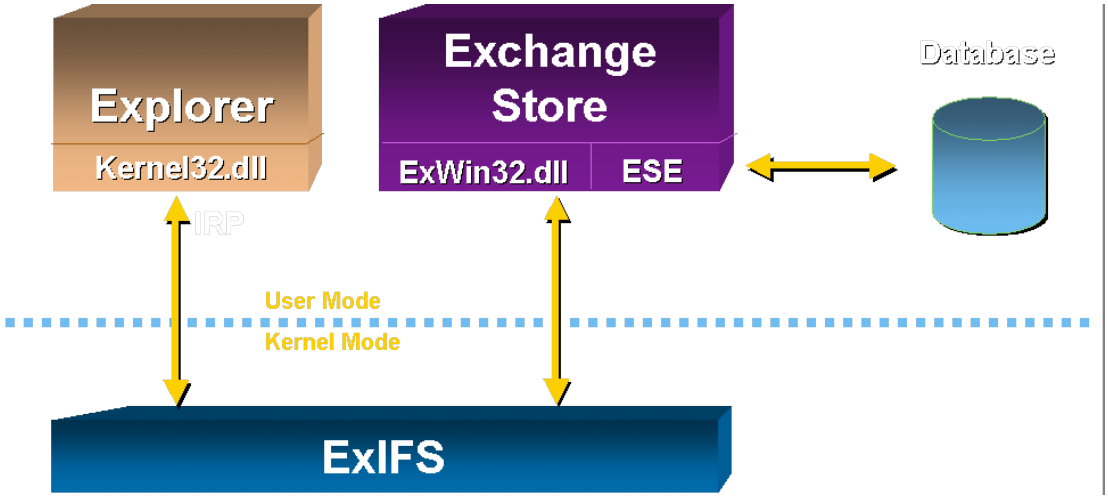
TODO: Better explanation needed, losing data is not a problem as the client had never released the handle – need to word this better

Filesystem semantics

Exchange's IFS reflects WIN32 file calls into the store. The WIN32 filesystem API (i.e. CreateFileEx, ReadFileEx etc.) that is known and loved by millions of developers worldwide can now be used to get at data in Exchange.

Calls like FindFileFirst can look in a public folder for a list of messages. You could map a drive (net use ?) to your own mailbox and use the traditional command line (dir, copy and move etc.) to access your inbox (if you wanted to)

Essentially IFS exposes the contents of an Exchange database as ordinary files.



The diagram shows that as well the interaction with the store achieved through ExWin32.dll, ExIFS.sys also supports all the file system related functions that are exported from kernel32.dll.

Transactions

A transaction is a series of operations that are treated as atomic (indivisible). So the operations in a transaction are either all done and permanently saved, or none of them are done.

As an example we can consider the operations involved when we do something very simple like move a message from the inbox to the deleted items folder.

The message has to be deleted from one folder, it has to added to another folder, and the folder properties need to be updated - number of items, read / un read count etc.

We do not want some of these operations to happen and not others. Again we are back to the design principle 'What happens if we crash?'.

If we crashed in the middle of this series of operations and only some operations finished we might end up with two copies of the message, worse no copies, or folder properties (such as item count) that were inconsistent with the folder contents.

To prevent this happening we bundle operations inside a transaction, and then ESE handles this idea of committing a transaction.

ESE will ensure that none of the operations are permanently applied until the transaction is committed, and when the transaction is committed all of the operations are permanently applied.

So if we crash before the commit we do not see any part of the transaction, and if we crash after the commit we see all of the transaction.

A standard approach to addressing the question of reliability is the notion of ACID (atomic, consistent, isolated, durable) transactions.

ACID

Haerder and Reuter in 1983 (published in Gray and Reuter's book 'Transaction Processing: Concepts and Techniques'*) defined the four properties a transaction must have:*

Atomicity

A transaction's changes to the state are atomic either all happen or none happen. These changes include database changes, messages and actions on transducers.

Consistency

A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.

Isolation

Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

Durability

Once a transaction completes successfully (commits), its changes the state to survive failures.

The Version Store

The version store gives ESE the ability to track and manage current transactions; this allows ESE to implement Isolated and Consistent transactions.

The version store keeps an in-memory list of modifications made to the database. This list has several uses:

Rollback

If a transaction needs to rollback it looks in the version store to get the list of operations it performed. By performing the inverse of all the operations the transaction can be rolled-back

Write-conflict detection

If two different sessions try to modify the same record the version store will notice and reject the second modification

Repeatable reads

When a session begins a transaction it always sees the same view of the database, even if other sessions modify the records it is looking at. When a session reads a record the version store is consulted to determine what version of the record the session should see

Deferred before-image logging

A complicated optimization that lets us log less data than "other" database engines

JET_errVersionStoreOutOfMemory

If we have very long-running or hung transactions the version store can grow quite quickly, and ultimately it is possible to see version store out of memory errors.

Version store out of memory errors (-1069) where once a symptom of the lost Long Value root bug I talked about earlier, but if you get them now on a recent Service Pack you probably need to be talking to PSS to try and diagnose the problem.

The version store and Multi-user considerations

Once a transaction has begun ESE will guarantee the session will see a single consistent image of the database as it existed at the start of its transaction plus its own changes.

However other sessions may also be modifying the data and committing their transactions; these changes are invisible to any transaction that started prior to the commit.

A user may modify a record only if that user is looking at the latest version; otherwise the update fails with JET_errWriteConflict. Versions older than the oldest transaction are automatically discarded.

Note the version store in ESE guarantees total data consistency even for very long-running transactions without locking out any other sessions. However it does not in itself guarantee serializability of transactions. Explicit locks are supported for this purpose.

Snapshot Isolation

Snapshot Isolation (SI) is a concurrency control algorithm that was first described in 'A Critique of ANSI SQL Isolation Levels' (O'Neil et al.). SI is implemented by Microsoft Exchange through the use of repeatable reads.

Repeatable reads

Repeatable read gives us an isolation level where, once a session begins a transaction it sees the state of the database as it was when the transaction began, regardless of modifications made by other sessions. You might contrast this with dirty reads where all modifications made to the database are instantly visible. Repeatable reads are implemented using the version store, with an in-memory list of modifications made to the database it is fairly easy to work out what view of a record any particular session should see.

Serializable transactions

Repeatable reads do not indeed produce serializable transactions. One example of this might be having one session read a value from a record and write the value to a second record while another session reads a value from the second record and writes it to the first:

```
Original Contents:
      Record1:    "Hello there"
      Record2:    "Goodbye"
Modifications:
      Session1: Begin transaction
            (Session1 will now always see the original
            contents of Record1 and Record2 unless Session1
            modifies the records)
      Session2: Begin transaction
            (Session2 will now always see the original
            contents of Record1 and Record2 unless Session2
            modifies the records)
      Session1: Read Record1, write contents to Record2
      Session2: Read Record2, write contents to Record1
      Session1: Commit transaction
      Session2: Commit transaction
Final Contents:
      Record1:    "Goodbye"
      Record2:    "Hello there"
```

This produces a non-serializable sequence of transactions! There is no possible non-interleaved ordering of the operations performed by Session1 and Session2 such that the same set of database modifications are made. For example:

```
Original Contents:
      Record1:    "Hello there"
      Record2:    "Goodbye"
Modifications:
      Session1: Begin transaction
            (Session1 will now always see the original
            contents of Record1 and Record2 unless Session1
            modifies the records)
      Session1: Read Record1, write contents to Record2
      Session1: Commit transaction
      Session2: Begin transaction
```

```
                    (Session2 will now see the modifications made by
                    Session1)
             Session2: Read Record2, write contents to Record1
             Session2: Commit transaction
      Final Contents:
             Record1:    "Hello there"
             Record2:    "Hello there"
```

Conversely, doing the modifications of Session2 first would mean that both records would end up containing "Goodbye".

Having non-serializable transactions will not corrupt a database at the ESE level. Recovery will redo/undo non-serializable transactions with no problems. It is possible though for non-serializable transactions to mess up database consistency at the application level. The store avoids any inconsistencies caused by repeatable reads by internally locking the objects that it is manipulating and not writing any code that could generate non-serializable transactions.

(In an all-purpose database system you could avoid this problem by "read-locking" records that you wanted to look at so that no-one else could modify them. That would mean that you would be unable to perform the non-serializable transactions in the example because the attempt to write to Record1 would cause an error because it was read-locked by Session1. ESE does indeed support read-locks but they aren't needed or used by the store. There is also the concept of write-conflicts in ESE. A write-conflict occurs if two concurrent transactions attempt to modify the same record. If that happens the second transaction will receive an error and be unable to update the record; the only way that the same record can be modified by different transactions is if one transaction committed before the other transaction began.)

Logging/Recovery

We know that the version store helps us track transactions while the system is running, but what happens when the system crashes? We lose the memory based version store.

Well to help us in this situation we use transaction logging.

We don’t want

So what happens if we crash? We don't want the data B-trees to be broken; we don't want index B-trees to be inconsistent with their data B-trees and we don't want partially committed transactions.

We do want

We do want a solution that performs well; we don't want the job of keeping a grip on consistency to be a drag on performance.

Theory

ESE uses the write ahead logging protocol (WAL). WAL gives us atomic and durable transactions, lets see what this means:

First

When ESE modifies a page it first modifies a cached version of the 4k-database page, we do not write it immediately to the database disk.

Second

So to secure data (make it durable) all we have to do is append that data sequentially to the end of the log, this form of sequential disk I/O to the log disk (especially if no other process is using the log disk) is relatively cheap and quick.

Third

The writes to the database disk we can defer, we can batch them up and write them to the disk using a background thread, preventing us from having any direct effect on database performance.

Scatter/Gather I/O

The performance of reading and writing to the database disk is improved by using Scatter/Gather I/O functionality that was added to Windows NT 4.0 Service Pack 2.

Win32 API calls ‘ReadFileScatte’r and ‘WriteFileScatter’ allow ESE to issue a single read or write from more than one buffer in virtual memory to a contiguous area of the database file on disk so ESE can take a large collection of pages that aren’t necessarily near each other and write them out in one large I/O and get the benefit of a large I/O, without having to do a lot of expensive bookkeeping to find pages that are next to each other.

Elevator Algorithm

If the lifts in a large tower used a shortest-seek-first algorithm, at busy times, they'd all head for the nearest floor to be visited, and end up mostly hovering around floors 5 and 6, with the occasional foray to higher and lower floors. Although total lift movement would

be greatly decreased, over a first-come-first-served algorithm, fairness would decrease (and average journey time would probably increase dramatically).

For this reason, lifts as does ESE use the so-called elevator algorithm. In this scheme, the ESE (or lift car) moves across the disk (or building), satisfying all the requests in one direction, before turning around, and satisfying the requests in the other direction... and so on.

Requests for I/O are given by the buffer cache to ESE which constructs lists of requests ordered by the elevator algorithm, which sorts the lists to make accesses faster and more efficient.

So, where is my mail?

The big implication of write ahead logging is that when we are running we have data in the log files that is not in the database.

Modifications to the database can remain in the memory buffer cache for many seconds.

It is completely possible to receive and read mail that is still in the database cache and has yet to make it to the database disk.

Consistent or Inconsistent

Each database has a flag in its header that shows either as 'consistent' or 'inconsistent'.

If a database is shut down cleanly and all of the data in the buffer is flushed to disk then the flag is set to ‘consistent’, this is the only time a database is marked as ‘consistent’.

If the database is running then the flag is always set to 'inconsistent' - there is data in the log files that is not in the database.

If a stopped database is inconsistent it is important to understand that this does not means it is corrupt or broken in some way - it just means the database was not shut down cleanly.

If a stopped database is inconsistent there is data in the log that is not in the database, so we need to run recovery to make the database consistent (i.e. play the data from log into the database).

Therefore what we always have to remember is that if a database is inconsistent then there is data in the log files that is held nowhere else - so the log files are critically important.

ESEUTIL /MH will indicate whether a database is consistent or not.

Log files

Conceptually, we could just have one huge log file, but that would be difficult to manage.

So we break the log into multiple files each file is exactly 5Mb, we call them generations. The hexadecimal generation number is used in the name of the log file.

The highest generation on the disk is always edb.log. When it is full we rename it using the generation number and create a new edb.log.

There is nothing special about edb.log other than that it is the current open log file that ESE is writing to, all of the log files on disk used to be called edb.log at one time.

If you see edbtmp.log on your disk for anything longer than an instant then it probably means the server crashed during log creation.

EDBTMP.log

EDBTMP.LOG is created as part of the ESE procedure for creating a new log file safely. We always have to consider what happens if we crash. We must consider what happens with circular logging on and circular logging off:

Circular logging on

- *Advance the checkpoint*
- *If the checkpoint is in the oldest log file, act as though circular logging is **not** on*
- *Rename the oldest log file to EDBTMP.LOG*
- *Delete all log files older than the checkpoint (except the one renamed to EDBTMP.LOG)*

Circular logging off

- *Create EDBTMP.LOG*
- *If creation fails because the disk is full, rename RES2.LOG to EDBTMP.LOG*
- *If RES2.LOG doesn't exist, rename RES1.LOG to EDBTMP.LOG*

Then

- *At this point we have an EDBTMP.LOG that is the appropriate size and ready for use*
- *Initialize the header of EDBTMP.LOG with the appropriate generation, signature and timestamps*
- *Rename EDB.LOG to EDBXXXXX.LOG*
- *If we crash at this point JET will notice that there is no EDB.LOG and will rename EDBTMP.LOG to EDB.LOG if EDBTMP.LOG doesn't exist recovery will fail*
- *Log records are not written to EDBTMP.LOG until is it renamed to EDB.LOG*
- *Rename EDBTMP.LOG to EDB.LOG*

Why?

Why do we do this? We never want an invalid EDB.LOG! If we simply created EDB.LOG we would be in a situation where, if we crashed before fully initializing the header, EDB.LOG would be filled with zeroes/garbage and we wouldn't be able to determine if we had a corrupted log file (recovery should fail) or a freshly created EDB.LOG (recovery should succeed). By using EDBTMP.LOG we are never in a situation where EDB.LOG is not a valid log file.

RES1.log and RES2.log are created during initialisation. They are used if we run out of disk space on the log drive. This allows a graceful shutdown.

What is in a log file?

A log file can be divided into 2 parts: the header and the data.

Header

The header contains the log generation; log signature, time stamp, and then some information about the databases - paths and signatures of the databases.

And it's important to remember this: a log contains hard coded paths, drive letters and full paths to the databases

Since Exchange 5.5 Service Pack 1 you can dump the log file header using ESEUTIL /ML

```
Initiating FILE DUMP mode...
      Log file: edb.log

      lGeneration (5)
      1 C:\exchsrvr\MDBDATA\PRIV.EDB
      Signature: Create time:11/3/1997 15:35:8 ...
      2 C:\exchsrvr\MDBDATA\PUB.EDB
      Signature: Create time:10/9/1997 8:6:15 ...
```

Above is an edited version of what you'd see if you dumped the log header. The big thing to note here is that we have hard coded paths to the databases.

Remember this - because when we talk about recovery you'll see directly how this affects what you can and cannot do.

Basically (because the paths are hard coded) if you move a database to a different location recovery cannot run.

When we run recovery we open the log file and look for the databases that might need to be recovered at the paths specified in the log. If the databases are not at the paths specified recovery stops.

And if you look at the situation with Exchange 2000 where we now have multiple storage groups and each storage group can have multiple log files you should be able to see that doing it this way is a good design decision and prevents dangerous things being done with log files on a server.

Log Records

After the header the rest of the log file is log records. The log records have a structure, they contain instructions such as BeginTransaction, Commit and Rollback and they contain the binary data that is being written to the page.

It is important to stress and understand that each record is at a very basic binary level, lets take a look at the sort thing we record...

```
Begin     (8)
Replace   27223(8,[1477:6],8,8,8)01 00 00 00 70 03 00 00
Delete    27150(8,[992:0])
Insert    27224(9,[1095:7],255)7F 14 2F 6F A8 1C ...
Insert    27225(5,[702:8],255)80 D7 74 C9 68 6C ...
Insert    27226(8,[696:1],255)80 94 26 BC B5 9B B5 ...
```

```
Insert    27227(8,[735:8],255)80 D7 74 C9 68 6C 17 ...
Commit    (8)
```

Above we see the way a transaction is recorded in the log

The session number corresponds with a thread doing some work, and here we see session 8 recording a complete transaction in the log: begin, replace, delete, insert, insert, commit

And for each of these operations we record the physical location - page number (1477) and offset (6) - that is being modified, and the raw data that gets written (01 00 00 00 70...).

We can see in this example and as is usually the case, multiple sessions (5, 8 and 9) and therefore multiple transactions are interleaved in the log file

You will also notice each log record has a timestamp, this timestamp is an ever-increasing number unique to each database, internally we call it dbtime, I will explain how we use this timestamp later.

Logfile Checksumming in Exchange 2000

We know that the logs protect the database from permanent corruption, so what protects the logs?

Each record in a log file has a checksum. The LRCK (Log Record ChecKsum) can cover a range of data both before and after its position in the log file.

The LRCK

A Log Record ChecKsum is a structure that contains the following fields:

```
le_cbBackwards     - Count of bytes previous to LRCK position to checksum
le_cbForwards      - Count of bytes subsequent to LRCK position to checksum
le_cbNext          - Count of bytes to the position of the next LRCK
le_ulChecksum      - The Checksum
```

The checksum is produced by XOR'ing every 4 bytes in the range (backwards and forwards), all fields in the LRCK itself, except for the le_ulChecksum and the generation number of the log file.

The generation number of the log file is included when calculating the checksum so when circular logging is enabled when we re-use an old log file we won't interpret the old records as valid anymore.

Each checksum is read verified during backup and recovery. If a checksum fails during one of these operations the operation will stop, logging an event in the application event log similar to:

```
Event ID:         463
Source:           ESE98
Category:         Logging/Recovery
Description:
```

```
Corruption was detected while restoring from backup log file c:\
exchsrvr\mdbdata\edb.log. The failing checksum record is located
at position 8948. Data not matching the log-file fill pattern
first appeared in sector 8949.
```

Two types of LRCK failure

During recovery if a LRCK fails to match the data it covers it can be for one of two reasons, either a torn write at the end of the current log file (edb.log) or data damage somewhere else in the log file sequence, ESE handles these two situations differently.

Torn write

Before ESE can act on a failed LRCK it must first work out where the data damage is. To enable ESE to do this each log file is filled before use with a data pattern known as the 'log-file fill pattern', if ESE is replaying transactions in edb.log ESE looks for the recognisable fill pattern beyond the LRCK range. If the fill pattern is the only data found beyond the point of the failed LRCK range to the end of the log file then ESE can safely make the decision that the invalid data was caused by a 'torn write' at the end of the log file sequence and that there are no log record beyond this point.

In the case of a torn write ESE will roll back the operations of all uncompleted transactions and place a new LRCK with le_cbBackwards = 0 at the place specified in the le_cbNext of the last LRCK. Recovery will complete successfully and new transactions will start after the new LRCK.

ESE will log an event to indicate it has hit this condition:

```
Event ID:         461
Source:           ESE98
Category:         Logging/Recovery
Description:

A torn-write was detected during hard recovery in log file c:\
exchsrvr\mdbdata\edb.log. The failing checksum record is located
at position 8948.
```

Or:

```
Event ID:         462
Source:           ESE98
Category:         Logging/Recovery
Description:

A torn-write was detected during soft recovery in log file c:\
exchsrvr\mdbdata\edb.log. The failing checksum record is located
at position 8948.
```

NOTE: Please see section 'Hard vs. Soft recovery' for differences between these two types of recovery.

Data damage

If however ESE is either not replaying transactions from edb.log or finds data other than the recognisable log-file fill pattern in edb.log beyond the failed LRCK data range there must be more records in this log file.

In this case data that the disk subsystem had confirmed to be successfully written to disk is now no longer not matching its checksum.

Since ESE cannot skip transactions the Administrator will need to:

TODO: Have played any transactions from the corrupt log in to the databases? I.e. did we check the log file upfront?

TODO: Are we consistent when we stop at this point?

If no and yes to the above questions

- check for transient checksum error in the failing log, if the data in the log proves to be good then we can perform recovery again
- check for transient checksum error in the failing log, if the data in the log proves to be bad then we can delete this log and all subsequent and mount the database.

OR:

If no to the second question

- check for transient checksum error in the failing log, if the data in the log proves to be good then we can perform recovery again (eseutil /r (soft), eseutil /cc (hard))
- check for transient checksum error in the failing log, if the data in the log proves to be bad we have to restore...

ESE will log an event to indicate it has hit this condition:

```
Event ID:         465
Source:           ESE98
Category:         Logging/Recovery
Description:

Corruption was detected during soft recovery in log file c:\
exchsrvr\mdbdata\edb.log. The failing checksum record is located
at position 8948.0. Data not matching the log-file fill pattern
first appeared in sector 8949
```

Or:

```
Event ID:         464
Source:           ESE98
Category:         Logging/Recovery
Description:

Corruption was detected during hard recovery in log file c:\
exchsrvr\mdbdata\edb.log. The failing checksum record is located
at position 3423.0. Data not matching the log-file fill pattern
first appeared in sector 3424.
```

NOTE: Please see section ‘Hard vs. Soft recovery’ for differences between these two types of recovery.

What is a Torn Write?

A write to the disk subsystem is done in block multiples of the disk sector size; the disk sector size today is usually 512 bytes.

During normal operation a block write to the disk subsystem will result in either a success (all of the data can be considered to have been committed to disk successfully) or failure (none of the data can be considered to have been written successfully)

A torn write occurs when an application requesting a block write is not told whether a write was a success or failure, leaving the application with out knowledge of the state of the disk sectors that made up the block write.

Example of situations when a torn write will occur:

- Power failure whilst in the middle of writing the sectors that makes up a block I/O request.

- After a disk subsystem has issued a success for a block I/O write request a failure to pass all the modified sectors between a disk subsystem cache (either controller or disk) and the physical disk itself.

- A breaking of a physical disk mirror whilst the disk subsystem is still receiving block I/O write requests will result in torn-writes on the disk broken from the mirror

ESE uses a range of techniques to recover from torn-writes, in the log manager we use the log-fill pattern. In the log and database manager ESE also uses shadowing techniques

TODO: Explain shadowing

Manual log checksum verification

The log file checksums can be manually verified using eseutil /ml, this performs a quick (i.e. less than a second) integrity check verifying each checksum.

```
.
.
.
Integrity check passed for log file: ..\mdbdata\edb.log

Operation completed successfully in 0.906 seconds.
```

Modifying the database

The first step in performing an update is to fetch the 4k page we want to modify off the disk. ESE then creates a new log record, gets a new timestamp, and writes the timestamp to the page in memory and the log record.

ESE then goes ahead and modifies the page, and records the modification made in the log record.

Now we don't actually write the log record straight to disk - remember we're just running somewhere in the middle of a transaction here.

What we do is link the log record with the database page in such a way that if the buffer manager tries to write the page to the database disk (say we come under memory pressure) then the log record must get written to the log first.

When the transaction commits we make sure at this point that all log records associated with that transaction are written to disk. After the commit completes we have saved every change associated with the transaction in a 'safe' place.

A 'lazy' transaction will not force all the log records to disk when it commits. This means the transaction isn't really durable. Exchange users lazy transactions to improve performance, this is only done when data integrity is not threatened. For example the information store uses lazy transactions when creating a restriction or search folder on a table.

When just one piece of mail arrives on an Exchange server there are literally dozens of low-level operations that have to be performed, all of which get recorded in the log.

For a example, a message arrives - the inbox must be updated, the table that will store the message must be updated, all the indexes on each of those tables must be updated, so there is a lot of individual changes that make up the transaction that are all going into the log.

In addition the store is doing things for multiple users at the same time, and all these operations are interleaved in the log. The result of all this is that you can't just dump a mail item out of a log file - if you do dump a log file the best you are likely to manage is to locate a few fragments, interleaved with a lot of data.

The message here is really that the contents of log files are pretty much opaque from the users' point of view - they work the way they are designed to work and you really can't do anything else with them.

Recovery

Recovery is the process that runs against an inconsistent database to make it consistent.

Now let's recap; we know what an inconsistent database is, it is a database that does not yet have all the data in the log played into it, for example this would happen if the server had crashed.

What we have to do here is update the database with the data that is only recorded in the log (i.e. has not made it to the database) - so we read the log and replay the operations into the database.

To do this we use the timestamp I spoke about earlier…

```
Begin     (8)
Replace   27223(8,[1477:6],8,8,8)01 00 00 00 70 03 00 00
...
```

To take this example here; during recovery we would read page 1477 off the disk and look at its timestamp.

If the timestamp on the page is the same as the timestamp on the log record then we know that this operation has been committed to the database, so we skip to the next log record.

If the timestamp on the page is older than the timestamp on the log record we know we have an operation that never made it to the database disk, so we play the modification into the database.

Each time we do this - each time we replay an operation - we use the version store to keep the version of the page before the change.

This means when we get to the last operation at the end of the last log file we can rollback uncommitted transactions, using the version store.

We call this replay of transactions, 'physical redo and logical undo'.

Recovery runs automatically when ESE starts. Any application that initialises ESE will cause recovery to run, whether it is the Information Store, ISINTEG (the Information Store Integrity checker) or some other utility.

Recovery runs on the log files, we have seen that the log files contains hard coded paths to the database - if the databases are not where recovery expects to find them then ESE will return a -550 error and stop.

The only way to move an inconsistent database is to restore an online backup to a new location, in this case the mapping from old to new location is in the RestoreInProgress key that is used by the backup API, and a different type of recovery runs and checks for a mapping.

The Checkpoint (EDB.CHK)

If we look at the recovery process we can see an opportunity for optimisation.

If we go back far enough in the log we will reach a point where every log record older than 'here' (we call it the checkpoint) references a modification that has already been flushed to the database disk.

Because recovery only acts on log records that have not been played into the database there is no need to look at log files older than this checkpoint.

So the checkpoint answers the following question: if I crash now where do I need to start recovery from? If we have hundreds of log files, starting recovery from the checkpoint represents a significant optimisation.

The checkpoint also allows us to implement circular logging. Circular logging deletes log files older than the checkpoint, these are log files that are not needed for recovery to complete. Although circular logging doesn’t stop recovery from succeeding you must recognise that it does stop you from rolling forward log files after restoring an online backup.

Implications of physical logging.

Now that we all completely understand physical logging lets look at some of the issues it is possible to run into that can cause problems in Exchange 5.5's ESE97.

The fact that we do all our logging at a physical (binary) level has major implications, and the biggest of these is that the database must be in the correct physical state for recovery to run.

Delete edb.chk

What happens if when running recovery we encounter an operation that tells us to delete a record, but the record was never added to the database in the first place? It can happen like this: in log generation 9 we insert a record, and in log generation 10 we delete it.

```
EDB00009.log: Insert    184976(9,[1095:7],255)…
EDB0000a.log: Delete    217345(8,[1095:7])
```

If we remove log generation 9 (edb00009.log) from the log directory then run recovery, recovery will fail because the checkpoint will tell ESE that log generation 9 should exist (we assume the checkpoint is at generation 9 or earlier).

However if we delete the checkpoint then recovery will run and simply act on what it finds in log generation 10. (ESE would not know log generation 9 is needed for recovery to run.)

When it comes to the operation in log generation 10 in this example it will read page 1095 off the disk and look at the timestamp on the page.

The timestamp on the database page might be the same as the timestamp on the insert operation from log generation 9, and in this case we're going to be OK (the delete operation in log generation 10 will remove the data that was inserted in log generation 9).

It is just as likely however that, the timestamp on the database page is older than the timestamp in the operation logged in log generation 9, in this case this operation was never flushed to the database disk.

In this case the timestamp on the database page is older than the timestamp on the operation in log generation 10 so we go ahead and apply the change. We delete whatever binary data happened to be at this location, without any knowledge of what that data actually is.

We have corrupted the database. And we've done it in a way that may not be detected straight away, in fact it may be days or weeks before you spot the problem, and start blaming Exchange for corrupting you database.

The really important idea to recognise is that the checkpoint prevents this type of logical corruption to your database from happening.

Delete edb.log

We know now that we should not delete the checkpoint (edb.chk). So looking at this from a slightly different angle what happens if we delete edb.log?

At first glance you might think this is a safe action to perform, by dumping the checkpoint we can ascertain the checkpoint is a few files back so none of the records in edb.log could have made it to the database disk, we should be safe? No!

The problem is that it is very likely that some records in edb.log are for changes to pages that have been flushed to the database.

The checkpoint only tells us the point where everything has been flushed to the database; there will always be plenty of records after the checkpoint that have also been flushed.

What will happen here is that you will end up with partially committed transactions - for example any transaction started in edb.log will not get rolled back - the result could either be a relatively trivial partially moved message, or a major internal inconsistency in one or more B-trees.

Write-back caching

If you run an unprotected write-back cache against the log drive – in other words a cache where you will lose data if the power goes down, then you will run into the same problem.

ESE assumes that if a write to the log disk completes then the data is made durable on the disk and it is free to flush the corresponding page to the database disk.

If later it turns out the data never made it to the log disk then we have this same problem, recovery is unable to roll back everything that it ought to roll back, and the database is corrupted.

This does not mean that write back cache is outlawed, but it does mean you need to make sure the data in the cache is properly protected with battery backup, error checking and correction and sound operational procedures.

Here is an example in a little more detail of what can go wrong. This session makes a bunch of changes, in fact a whole transaction, within edb.log

```
EDB.log:
      Begin     (8)
      Replace   27223(8,[1477:6],8,8,8)…
      Delete    27150(8,[992:0])
      Insert    27226(8,[696:1],255)…
      Insert    27227(8,[735:8],255)…
      Commit    (8)
```

You can delete edb.log and rename the next highest log generation to edb.log, and recovery will run. But the problem is we just have no way of knowing which of the pages relating to this transaction got flushed through to the database disk, and recovery won't touch any of them. We deleted the log file recovery will ignore the transaction.

The likelihood is that some but not all of these modifications have been flushed to the database, and in this case we have guaranteed corruption of the database even through recovery appears to run OK.

The only question here is how bad the corruption is and will the database run a few days, a week or just crash straightaway?

Exchange 2000 will prevent mistakes of this nature from happening, but the safeguards to do this do not appear in Exchange 5.5 or any earlier version, so we need to understand the risks of making mistakes.

The fact is that if recovery doesn't run through to completion on its own then it is almost never a good idea to hack the log files to get it to run.

What can you do in this situation? The only time you can safely delete one or more log files from the sequence is when you have restored from an online backup, in this case you should be able to roll forward all the way up to the file immediately before the one you deleted.

Recovery Error Detection in Exchange 2000

As we have seen in Exchange 5.5 playing the wrong log files into a database will corrupt the database, this corruption will be a logical corruption and it's side effects can vary from the subtle to the catastrophic.

ESE98 sees four areas of additional functionality in the ESE log manager that will prevent the wrong log or the wrong log sequence from being played into a database.

All of this extra functionality in the log manager is there to prevent Administrator error

Before and after timestamps

Each log record now has a before and after timestamp, this timestamp is called dbtime. Dbtime is an ever increasing number specific to each database, each page contains the dbtime of the time the last change was made to the page.

Each log record contains the dbtime that should be on the page before the operation is played into the database and the dbtime to place on the page after the operation is played in.

When a operation is due to be played into a page in the database if the log record before dbtime is higher than the dbtime on the page then ESE will return JET_errDbTimeTooOld (-566), if the log record’s dbtime is lower than the dbtime on the page ESE will return JET_errDbTimeTooNew (-567), in either case recovery will fail therefore preventing the database from becoming logically corrupted.

Impossible to run soft recovery on restored database

ESE98 makes it impossible to run 'soft recovery' on a database that needs 'hard recovery', any attempt to do so will result in ESE returning JET_errSoftRecoveryOnBackupDatabase (-544).

Hard recovery is the process that takes place during a restore of a database, i.e. the patch files and log files are played into the database.

Soft recovery is run when a database is initialised and the database is in an 'inconsistent' state (i.e. after a crash or sudden power down). Soft recovery is not aware of patch files.

Soft recovery can be forced on a database by running eseutil /r, this unfortunately is often done by Administrators who are restoring a database and notice (by dumping the header) that it is in an inconsistent state, which indeed is should be… running eseutil /r, performs soft recovery and marks the database as 'consistent' it ignores the patch file and logically corrupts the database.

Log Required

A new field (Log Required) has been added to the database header.

‘Log Required’ is the range of log files needed for an inconsistent database to become consistent through the recovery process.

On initialising a database that is inconsistent ESE will verify that the log range specified in the database header is present in the logs directory and the logs are the correct generation (i.e. the database and log signatures are correct).

Recovery will stop and return JET_errRequiredLogFilesMissing (-543) if the log range specified in ‘Log Required’ is not present.

Database Mismatch

When recovery starts ESE will also notice if:

- Any of the databases at the time of a crash are no longer present at the start of recovery. We cannot allow this to happen because the checkpoint file will be advanced preventing the missing database from ever becoming consistent again.
- If a consistent database is trying to have log transaction played into it that is not the next transaction in the log sequence, this would happen if there is a gap between the last transaction played into database and the log transaction currently trying to be played into the database.

ESE will return JET_errAttachedDatabaseMismatch (-1216) if either case occurs.

Backup and Restore in Exchange 5.5

So if we take all we have learnt together, inconsistent databases, transactions, log files and we want to back up without taking the system down, we end up with the Exchange online backup strategy.

The patch file

What we are going to do here is take a backup of a database while it is being modified. There are three pieces to how this works: you have already heard about the databases and the log files, the extra piece used by online backup is the patch file.

During online backup, a page will be placed into the patch file if all of the following conditions are met:

- The page has just been flushed to disk.
- The page is the destination page of data movement (i.e. data is moving from one page to another because of a B-tree split or merge).
- The page either has already been copied to the backup set or will not be copied to the backup set (because it is beyond the pre-established backup endpoint i.e. the size of the database at the time the backup began).
- The source page of the data movement has not yet been copied to the backup set, but will be (ie. the source page is past the current copy point, but before the pre-established backup endpoint).

In all other cases, ESE can reconstruct the destination page because the data is either in the backup's log set or in the source page of the backed-up database.

If the system is in a reasonably quiescent state during the backup, there should be little, if any, patching involved.

Backup the database

To do all of this we need an Exchange aware backup program, like NT Backup or any of the 3rd party offerings available today.

The ESE API will copy the database in 64K chunks (adding pages to the patch where necessary) and hand the data to the backup program that will commit it to tape.

After we have backed up the database file we make a copy of the log files that are needed during the restore process to make the database 'consistent'

Log File housekeeping

Log files older than the log generation specified in the ‘checkpoint’ are no longer needed for data recovery because all the data in these log files is now safely in the database just copied to tape. The backup program will remove these files from disk preventing the build up of these un-needed log files

Restore the database

To restore a database we copy the database file, log files and patch file back to the source directory.

A restore in progress key is created, this key provides information about the log files (i.e. the low and the high log generation numbers) and also a mapping of a new database path, this allows us to move the database location of an inconsistent database.

When the service is started (i.e. the information store) it finds the Restore in Progress key, this is read and a different form of recovery is run, we call this hard recovery.

Hard vs. Soft recovery

The important difference in hard recovery is that the pages in the patch file are read in when needed. The other type of recovery described earlier, i.e. after a crash, we call soft recovery, soft recovery does not use the patch files.

This is where we can see a subtle type of database logical corruption you need to be aware of. Some people have become confused after restoring the files from tape – they see a patch file on disk, so they run ISINTEG –patch. But isinteg does not actually know about patch files, it initialises ESE in a way that runs the ordinary post-crash recovery (soft recovery), which never looks at the patch files, and you end up with a strangely corrupted database.

After performing a restore of an online backup in Exchange 5.5 one must always make sure to start the store before running any utilities, this will ensure that hard recovery runs and makes the database consistent in the proper way.

Backup and Restore in Exchange 2000

Support for backup and restore in Exchange 2000 is provided by two new dlls, eseback2.dll which interfaces with the service to be backed up/restored and esebcli2.dll which interfaces with the backup client be it NT Backup or a 3rd party vendors offering.

Exchange 2000 supports the online backup and restore of the Information Store, Key Management Service and Site Replication Service databases.

Storage Groups are the preferred entity of backup and multiple storage groups may be backed up or restored simultaneously.

What changed in Exchange 2000

The most obvious change in the restore process is that the Exchange service having data restored to it must be running.

The Database that is to be restored must be offline (i.e. if it was an Information Store database being restored, the database must be 'Dismounted' via the Administrator snap-in). Only the database(s) to be restored need to be offline

Exchange 5.5 saw the system attendant perform the restore recovery process. In Exchange 2000 the system attendant no longer plays a part in this area.

There is no longer any dependency on the registry, i.e. the Restore in Progress key is no longer used.

Instead a file (called restore.env) is written out to the temporary directory provided by the user during the restore process.

Client Setup

When Exchange is installed it adds a value to an NTBackup registry key to indicate where the Exchange client side .dll can be found.

NT Backup looks for this key each time it is started and will "Exchange enable" itself when present. (In Exchange 5.5 the whole NTBackup executable had to be replaced)

There is a new backup related key in Windows 2000 “FilesNotToBackup”. Applications that use their own backup methods (i.e. Exchange and Active Directory) must enter the files or directories that should be ignored by normal backups, for Example exchange will enter the mdbdata directories containing the information store files.

The Backup process

Exchange 2000 backup applications will use the following APIs to backup a set of databases and their log files. The calling sequence is as follows:

1. HrESEBackupPrepare() - To establish an RPC connection to the server application. Internally the server calls JetGetInstanceInfo() to get the layout of available databases to backup.
2. HrESEBackupSetup() - tell server which storage group to backup. Internally the server calls ErrESECBPrepareInstanceForBackup() which may disable shutdown.

3. For tape backup: Backup application stores the list of database names in the beginning of the tape (including display names and .stm file names)
    4. HrESEBackupOpenFile() – to open the database for read.
    5. HrESEBackupReadFile() – to read the database.
    6. HrESEBackupCloseFile() – to close the database.
7. Loop back to 4) until all database files (.edb and .stm) are read.
8. HrESEBackupGetLogAndPatchFiles() – get a list of log file names and patch file names.
    9. HrESEBackupOpenFile() – to open the log/patch file for read.
    10. HrESEBackupReadFile() – to read the log/patch file.
    11. HrESEBackupCloseFile() – to close the file.
12. Loop back to 9 until all log and patch files are read.
13. HrESEBackupTruncateLog() – to delete logs after backup.
14. HrESEBackupInstanceEnd() – to end the backup for the instance.
15. HrESEBackupEnd() – to disconnect from the server app.

Steps 5 and 10 (HrESEBackupReadFile) will checksum the database/log/patch data as it is returned to the backup application.

Log Truncation

ESE will not truncate the transaction logs until all databases in the storage group are backed up.

This is done so each database in the storage group can be restored from the last known good backup and have the complete transaction log sequence from the point in time of the back up to the current log generation on the disk, this continuous sequence of logs is essential for the transaction log roll-forward.

Because only mounted databases can be backed up, if a database is dismounted at the time of a backup (even if all databases in the storage group have been specified to be backed up) the database will ignored and the transaction logs will not be truncated.

```
Event ID:         951
Source:           ESE98
Category:
Description:

Unable to purge transaction logs because at least one database
(%3) is off-line.
```

ESE decides which log files to remove by taking the lowest number of either the log generation in ‘Checkpoint’ field in edb.chk or the ‘Log Gen’ for the ‘Current full backup’ in each database header. Once it knows this number ESE truncates the logs up to this generation.

Backup Granularity

Although any combination of databases in a Storage Group maybe backed up in any one backup set, it highly recommended that due to log file truncation the smallest unit of planned backup is the Storage Group.

Each Storage Group has it's own backup set, at restore time the Administrator may choose which elements (i.e. databases) should be restored from the backup set.

It is entirely possible to run concurrent back ups of multiple storage groups, so potentially if the hardware was available (lots of tape drives and some chunky SCSI cables) it would be possible to backup and restore at 25Gb/hr (typical tape speed) all 15 storage groups simultaneously, allowing a server to restore 1.5Tb in a 4 hour time window.

The Restore process

Exchange 2000 backup applications will use the following APIs to restore a set of databases and their log files. The calling sequence is as follows:

1. By reading from the head of the tape, the backup application will get a list of databases to restore; allowing the user to pick the databases from the list to restore.
   2a. HrESERestoreOpen() - if this is the first backup set to be restore, or:
   2b. HrESERestoreReopen() - if this is not the first backup set to be restored. These functions take a user specified temporary directory destination for the log and patch files.
3. HrESERestoreAddDatabase() – called once for each database to be restored.
4. Restore the database files from tape to the destination specified.
5. HrESERestoreOpenFile() - called once for each log and patch file to be restored
6. Restore the log and patch files to the user specified temporary directory destination
7. HrESERestoreSaveEnvironment () will save the restore environment as a file called restore.env (database mapping, log range, etc) to the temporary directory destination with the new information
8. HrESERestoreComplete() – Called once all the needed backup sets (differential/incremental and the full backup sets) are restored and the restore environment is saved.
9. HrESERestoreClose()

Restore Granularity

As already said it is possible at restore time to only restore the storage group or even the individual database needed. All other storage groups and databases remain unaffected. Therefore the only users affected by a restore are those on the database being restored.

Multiple sessions might be performing a restore at the same time. The only restriction in the number of concurrent sessions is a slight one.

The 16th instance

As I have already mentioned there may be up to 15 storage groups, ESE hosts each storage group internally as an instance.

ESE in fact supports 16 instances (one more than the information store can use). This last instance is reserved for the restore process that needs an instance itself to perform the hard recovery (transaction log roll forward) that starts when the backup application calls HrESERestoreComplete().

So if the server has 15 storage groups/ESE instances running and the Administrator wanted to restore two databases, one from one storage group, and one from another,

whilst not actually stopping a storage group/ESE instance (a storage group is stopped when no databases in the storage group are mounted) then the Administrator would only be able to perform the HrESERestoreComplete() part of the restore one at a time.

Administrator Actions

The Administrator actions involved in performing a restore:

To restore a database the service (the Exchange Information Store in this example) must be started, totally different to Exchange 5.5.

The database being restored must be dismounted before the restore is attempted.

Restore the backup sets including any incremental and differential backups. The Full backup (i.e. the databases) is restored last, when restoring the Full backup it is important the Administrator tell Exchange it is the last backup set to be restored by ‘ticking’ the ‘Last Backup set’ check box.

NT Backup asks for a temporary location, this is the location for the transaction logs and patch files that are restored from tape, additionally the 'restore.env' file is placed here (more on this later)

Before pressing 'OK' on the last backup set the Administrator must tick the "Last Backup Set" box, when this is ticked the "Mount Database after restore" box is available to be ticked as well.

I imagine ticking the "Last Backup set" box will be forgotten and fortunately rather than having to do the last restore all over again (which is the big one - the databases) it is possible to go through the motions of the restore again without actually specifying anything to restore, but just specifying the temporary location and ticking the box

The databases will replace any existing databases on the drive, but all other files will be restored a temporary location which the Administrator is asked to provide.

A file called 'restore.env' will appear in the temporary location. This file contains the detail that in Exchange 5.5 was stored in the Restore in Progress key.

This file is updated each time a backup set (i.e. an incremental or full) is restored and contains the databases restored and drive mapping between the backup and restore location and the log generation low and high numbers.

HrESERestoreComplete

Initialization

By ticking the "Last Backup Set" box, the Administrator is telling the service that all files are there to perform the "hard recovery"

The Backup application does this internally by calling the ESE function HrESERestoreComplete and logs the following event.

```
Event ID:         901
Source:           ESE98
Category:
Description:
```

```
Restore started from directory c:\RestTemp.
```

HrESERestoreComplete starts up a temporary instance (storage group) in ESE.

Log File Preparation

ESE verifies the log signature and generation sequence of the transaction log files:
If a log file signature does not match ESE will return
JET_errExistingLogFileHasBadSignature (-610).
If there is a gap in the log file sequence ESE will return
JET_errExistingLogFileHasBadSignature (-611).
If ESE returns any of these errors it will stop the recovery process before playing any transactions into the database, giving the Administrator chance to correct the problem without having to perform the restore again.

Once the log files have been verified ESE reads in the information contained in the patch file.

Log File Replay

First, the log files present in the temporary location are replayed. ESE then looks in the target storage group's live transaction log directory and replays the subsequent log sequence which starts after high log number from the temporary location.

Finishing the Restore:

The Store will mount the database if requested into the running storage group. The restore.env and the files restored to the temporary directory are then deleted.

The process that takes place after HrESERestoreComplete is started can as you imagine take a period of time, especially if the number of transaction logs to be replayed is large.

During this time the backup vendors have two options, they can either 'block' waiting for HrESERestoreComplete to complete and inform the Administrator something is happening or they can ask HrESERestoreComplete to return immediately (by passing in the flag ESE_RESTORE_COMPLETE_NOWAIT) and inform the Administrator to 'not turn the computer off' and look in the event viewer to ensure the restore completed successfully. ESE will log either a success or failure using one of the 2 events below

```
Event ID:         902
Source:           ESE98
Category:         Backup/Restore
Description:

Restore from directory c:\RestTemp ended succesfully.
```

Or:

```
Event ID:         902
Source:           ESE98
Category:         Backup/Restore
Description:

Restore from directory c:\RestTemp ended with error (%4).
```

Snapshot Backups

Snapshot backups are an exciting area that a few vendors have begun to implement in Exchange 5.5. Exchange 2000 is going to extend its backup API to support this form of backup natively.

What is a snapshot backup? It is the ability for the hardware to make an instant copy of the database files by performing a 'split'. The split is typically of a logical disk volume. The split volume is then a separate entity that can be independently manipulated and streamed to tape media at leisure.

Exchange 2000 will allow the vendors to split a logical volume containing Exchange databases (.edb and .stm) whilst the databases are still mounted and running without any noticeable effect to the Exchange users

The hardware needed to do this disk splitting is expensive and will probably be used by organisations wanting to host large volumes of users on few servers.

There is no need to stop Exchange when performing the snap of the physical disks.

Because the information store is still running the databases are snapped in an inconsistent state, i.e. there is still a need to perform hard recovery on a snapped backup.

The choice of granularity of backup is still unchanged, although as I have said the number of databases on a logical volume will typically dictate it.

Those vendors who have achieved snapshot backups in Exchange 5.5 have to manipulate the transaction log files (significantly deleting edb.chk to do a restore), there will be no need to do any such manipulation in Exchange 2000.

Microsoft is working closely with hardware and backup software vendors at the moment getting them going with the new APIs

The Snapshot backup application process

The Administrator chooses the database to backup (snap), the backup application decides the best way to split the volumes. At the moment there is still investigation in this area, it is likely there will be a backup set for each logical volume.

The backup application will follow the same steps as described previously in the section ‘The Backup Process’ - except steps 4 to 7 will be replaced with

4. HrESESnapShotStart() - The backup application tells ESE to quiesce the databases
5. The backup application issues the hardware command to split the logical volume.
6. HrESESnapShotStop() - The backup application tells ESE the split has finished, ESE is now allowed to release the pause on writes to the .edb database file.

The Backup program then copies the log files ESE says should be copied (these will typically be copied using traditional copying techniques i.e. not a snap)

Checking Physical integrity

Unlike a traditional backup ESE is unable to verify the physical integrity (verifying the database checksums) of the data during the backup, therefore it is up to the backup application to verify the physical integrity of the snapped databases by verifying the checksums in both the .edb and .stm file.

To verify the page checksums in the .edb file the backup application may use ESEFile that shipped in SP3 (see the ESEFile section at the end of this document)

However, verifying the checksums in the .stm file is going be interesting, because as said earlier the checksums for the pages in the .stm file are stored in the .edb file and we know the .edb file is running therefore in an 'inconsistent' state during and after the ‘snap’ (i.e. like any normal backup HrESERestoreComplete() needs to be called on it to make it consistent). Therefore ESE is unable to initialize the database to retrieve the checksum data.

This is an area being worked on currently, it is likely a file will be output during the call to HrESESnapShotStart(). This file will contain the checksums and offsets to the .stm file and a utility (maybe a switch in eseutil) will be available which will take this file as the input and verify the checksums.

If both the .edb file and the .stm file do successfully pass the checksum verification it is safe to say the backup was successful, therefore the backup program can ask ESE to truncate the log files (HrESEBackupTruncateLog).

Pausing Database Writes

As mentioned previously, HrESESnapShotStart quiesces the databases on the logical volume. To do this:

- Dirty pages (that is 4k pages that are newer in the memory buffer than the page on the database) are flushed to the database.
- Writes to the .edb database file are then stalled, however new transactions may still be committed to the transaction log files and pages will continue to be 'dirtied' in memory, this can continue until the buffer cache (dynamic buffer allocation) fills up.
- Writes to the .stm stream file will continue to be written out to disk (the .stm file), this will lead to torn writes during the snap (see section ‘What is a torn write?’ for more detail).

To overcome torn writes all the transactions in the logs during a snapshot backup destined to the .stm are unconditionally replayed, guaranteeing any 'torn writes' are over written.

Restore of a snapshot backup

The restore process of a snapshot backup is the same as a traditional restore except there is no need to restore the databases files (step 3 in the ‘Restore Process’ above), they are simply snapped back. Again the unit of restore will probably be the logical volume.

Space Management

Space in the database file

There are two types of space the database file: owned and available. Each table, index and long-value B-tree has its own list of owned and available pages. Available space is a list of pages that can be used to store new data. It is always a subset of owned space.

Owned space

Owned space in ESE is organized in a hierarchy. There are three levels to the hierarchy: the database root, the tables, and the indexes and long-values. The database root owns all the space in the database. Tables request chunks of the space which they then own (as well as the database root). In turn, index/long-value trees ask for space from the table that owns them. In order to reduce the number of requests and to avoid fragmenting space too much, requests for more space are made in chunks (normally 16 pages or 64K).

Available space

Available space is slightly different. A page can only be available to one level: either the database root, a table, or an index/long-value.

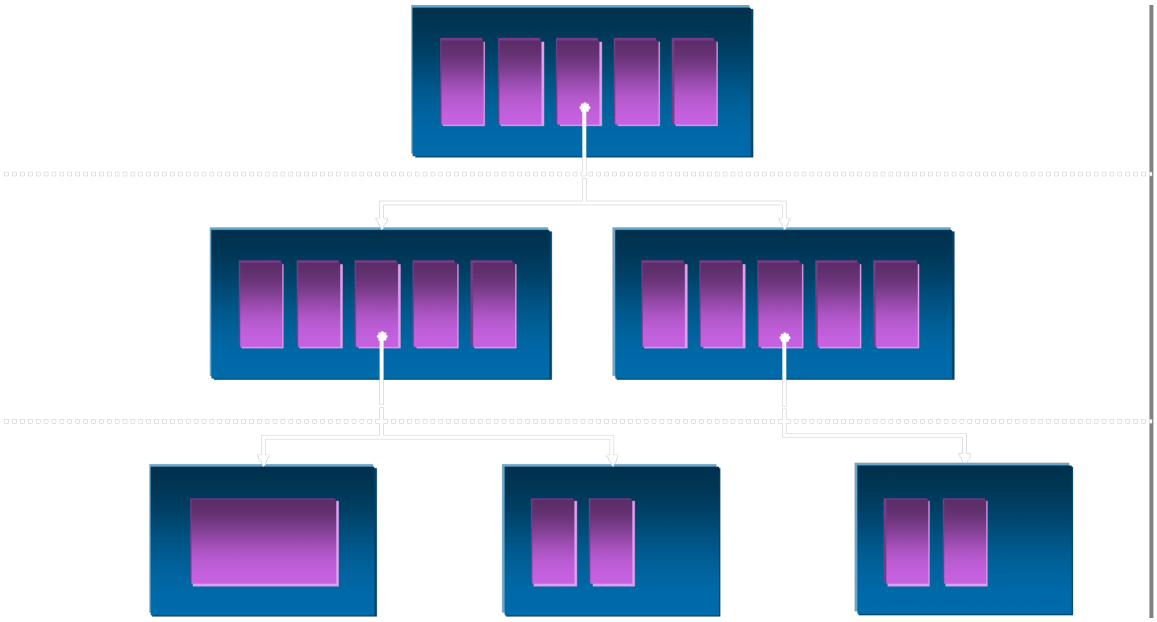
Why is my (4.0/5.0) database so big?

In Exchange 4.0 and 5.0 we did not reuse very efficiently. A page was only reused when it became completely empty

When a record was deleted we would look at the page next to the one we just deleted the record from, to see if it would be possible to fit the data from the two pages onto one page, and then free the spare page.

If it was possible we merge the data and free a page, but if it was not possible we just left things as they were.

In a messaging database most mail gets deleted, but because these are B-trees we are forced always to insert a new record in only one place. So over time you could end up with the situation where the database was 51% full – lots of free space that wasn't getting re-used. This was why it became a general best practice in Exchange 4.0 and 5.0 to schedule a regular offline defrag.



On the left, we have a page with one record, if we deleted this record the page would be completely free and therefore returned to the table as available space.

However the page in middle has two records leaving a 3rd of the page available, so does its adjacent page; Exchange 4.0/5.0 is unable to merge the available space.

As you can see this can lead to lots of small fragments of available space inside an Exchange 4.0/5.0 database, so in Exchange 5.5...

Online defragmentation

Online defragmentation is a new feature in Exchange 5.5. ESE traverses the pages at the bottom of each B-tree (the leaf pages) in the database looking to see whether it can merge strings of adjacent pages into one single page, freeing up pages and returning them to the table's available space. The locality and contiguity of related pages inside the database file is maximized where possible:

Online defragmentation runs as part of system maintenance, which by default is between 1am and 6am each morning. If ESE can't complete a full pass through the database it will remember where it got to and resume from this point when the next IS maintenance window comes round.

It is widely understood that Exchange 5.5 online defragmentation has meant there is no longer a need for scheduled offline defragmentation.

Offline vs. Online Defragmentation

Since Exchange first shipped it has also been possible to perform offline defragmentation of the .edb database file. These are the differences between online and offline defragmentation

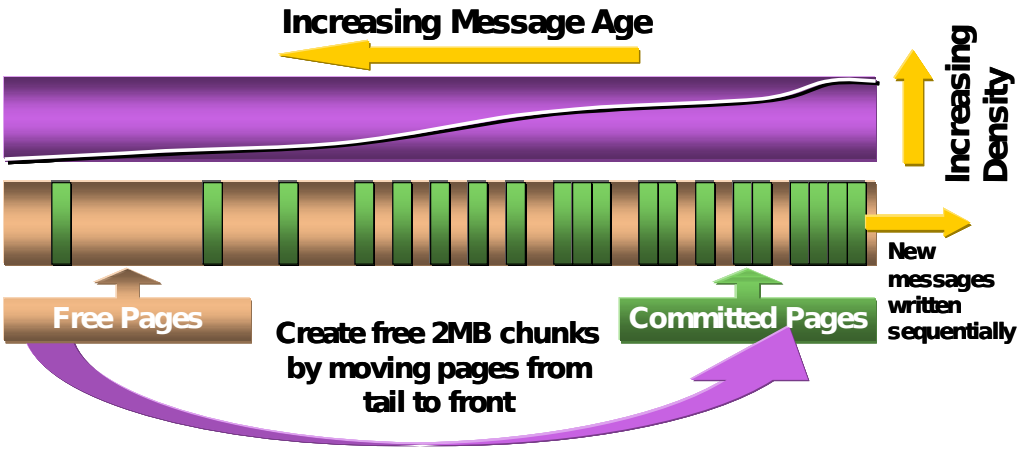
- Free space inside the database file (.edb) is not returned to the file system.
- Space trees are not removed
- Deleted column ids and long-value ids will not be reclaimed.
- Secondary indexes will be rearranged but not be rebuilt (i.e. if there is index corruption, it will not be repaired).

- No vertical merge support in the database file (.edb) (i.e. tree levels will not be collapsed).

From the user and administrator perspective the only difference that needs to be considered is the need to reclaim disk space. The only way this can be done is by running offline defragmentation (eseutil /d). Generally an Exchange 5.5 database will stabilize to a steady size state, therefore in Exchange 5.5 the only time there is a need to do an offline defrag is after some extraordinary behaviour such as a large move user off the server, or a looping rule that has expanded the size of the store.

Online Defragmentation in Exchange 2000

OLD has been extended to defragment the streaming file (known internally as OLDSLV). The streaming file is a simple flat structure (i.e. not a collection of B-trees).


Increasing Message Age
Increasing Density
New messages written sequentially
Free Pages
Create free 2MB chunks by moving pages from tail to front
Committed Pages

OLDSLV will pack the pages to the maximum possible density, and any free space at the end of the .stm file will be returned to the file system, the Information Store will log an event to indicate the amount of space that has been returned to the file system.

```
Event ID:          710
Source:            MSExchangeIS Private
Category:          General
Description:

Online defragmentation of streaming file 'c:\exchsrvr\mdbdata\
priv1.stm' has shrunk the file by 1,024,768 bytes.
```

In the release version of Exchange 2000 it should be possible to control the behaviour of OLDSLV to meet the individual server need by changing 2 values:

- Empty Threshold – The lower this percentage the less OLDSLV will pack the data therefore letting the .stm file grow larger leaving greater extents of free space allowing more I/O to be done in fewer requests
- Fragmented Threshold – The higher this percentage the greater OLDSLV will pack the pages in the .stm file therefore reducing the size of the file but increasing the number of I/O requests

The default values should be sufficient for most installations but where needed the trade-off between disk space and disk I/Os may be changed by altering these values

Free space discovery

After much customer feedback a new feature was added to Exchange 5.5 Service Pack 1. Administrators asked how much space would be regained from my Exchange database if I ran offline defragmentation (ESEUTIL /D).

```
Event ID:         1221
Source:           MSExchangeIS Private
Category:         General
Description:

The database has 9 megabytes of space after online defragmentation
terminated.
```

ESEUTIL /MS

ESEUTIL /MS will tell you the owned and available space for everything in the database. Note that the sum of the available pages gives the total number of available pages in the database (this number is printed at the end), while the number of owned pages in the database root gives the total number of pages in the database (i.e. do not sum the owned pages!).

The information in ESEUTIL /MS can be used to estimate the size of a database after a defrag. Here are some things to note:

- Not all the available space in the database will be reclaimed. As noted above, space requests are not made page-by-page but in larger chunks. It is possible (but rare) for a freshly defragged table to have as many as 15 pages available. This is a small error, but multiplied by the thousands of tables in a priv.edb or pub.edb it quickly adds up to many megabytes. If you run ESEUTIL /MS on a defragged database expect to see some available pages.
- You may get back more space than ESEUTIL /MS suggests. An offline defrag removes logically deleted records that have not been expunged from the database and packs the records together as well as possible.

To get a conservative estimate of the amount of space that will be reclaimed by an offline defrag do what the store does when it logs the free space: sum the available space in the database root, the attachments table, and the message table. You should find that the majority of free space is in those three places and that most of it is reclaimed by an offline defrag.

Problems with the way ESE97 allocates space

What is a space tree?

Each table in ESE has a space B-tree. The space tree is an 'on disk' B-tree structure containing the empty page numbers available to that table to store data (ESE collectively calls these pages 'Available space').

The page numbers of the empty pages are stored in the space tree as extents. An extent is a high page number and the number of contiguous empty pages before it. An example: if page 1000 and the 2 preceding where freed and added to the space tree, a single entry would be entered, this entry would be the extent 1000:3, i.e 998, 999 and 1000 are empty.

Typically the largest ESE tables in an Information Store database are the Messages table and Attachments table. These tables are represented in the ESE database by their data trees and typically very large long value B-trees; it is these long value B-trees that usually have largest space trees.

The MTS-ID is the search key used in the message & attachment tables data B-tree, the per server static first portion of the MTS-ID typically causes these B-trees to grow the middle of the B-tree, whereas the long value tables use an ever increasing search key value (Long-Value ID) causing these trees to grow the right of the B-tree.

When are empty page extents added to the space tree?

Pages extents are added to the space tree either:

1. During a normal deletion of data in a table (i.e. hard deleting a mail message) that causes page(s) to become completely empty. The extent of pages freed is added to the space tree

2. During normal deletion of data a page becomes small enough to be merged with it's adjacent right hand page.

3. During online defrag (aka OLD) pages are scanned to see if they can be merged with an adjacent page, if a page is freed by the merge the single page number freed is added to the space tree as a single page extent, this behaviour causes the space tree to grow significantly wider.

When is space allocated?

An ESE database stores data in pages, Exchange uses 4K pages. When a page becomes nearly full it is 'split'. When a page is split the data on that page is divided into roughly two halves, a page is requested from the space manager to place the latter half in.

How is space allocated?

In the build of ESE that shipped in Exchange 5.5 RTM when a page is requested the space manager will return:

1. If possible the closest page number that is higher than the page being split.

2. If a higher page number does not exist the closest page number that is lower than the page being split is returned.

3. If no page extents in the space tree exist a complex routine traversing the parent tables and possibly growing the database is performed.

Locality of pages:

The initial check to return the nearest page number higher than the page being split will return a page that:

1. Should be kept in a similar physical locality on disk,

2. Will probably be stored in a position beyond the current page in the same direction the disk heads are probably traveling.

Page locality is intended to allow the complex (and on my part little understood) ESE pre-read logic to perform a single disk I/O that can retrieve multiple pages reducing the expensive time cost of disk operations.

The initial problem:

There are two characteristics of the ESE space manager that caused a problem exhibited by a buildup of the 'receive queues' to the information store (typically the private store) and high B+ Tree moves nexts/prevs per sec (typically 50,000/sec).

1. Over a significant period of time after a large number of inserts & deletes the single page extents created by OLD cause the space tree for a large table to become extremely wide. Real world experience shows us a 60Gb database (that has never been offline

defragged) satisfying 2,000 average users will see tables beginning to exhibit very wide space tree after about 12 to 18 months of use.

2. The space tree becomes wide because the Exchange 5.5 RTM ESE space manager is unable to shrink the size of the space tree. The cause of the design problem becomes obvious when you consider the page number of a page freed from the space tree has to be added back to the space tree itself, this is a hard design constraint to overcome especially when every operation has to be recoverable from a crash.

Having a wide space tree is not too much of a problem whilst the tree contains a significant density of available page extents.

The problem is seen when there are few page extents available in a table's space tree that has previously been very wide. In this situation the space manager has to traverse the bottom of a space tree that contains mainly empty pages trying to satisfy space requests based on the rules described above (see: How is space allocated? above).

Resolution to the initial problem:

Exchange 5.5 Service Pack 2 abstracted access to the space tree by fronting it with an 'in-memory' cache containing only the available page extents ordered by page number. The space cache is built the first time we detect a space tree of sufficiently large depth that fragmentation in such a tree could potentially be a severe performance problem. The memory cache satisfies the requests using the same three rules (see: How is space allocated? above)

The memory cache is substantially faster at satisfying space requests for two reasons:

1. The memory cache is faster than access to the 'on-disk' space tree (memory access is approx. 200,000 times faster than disk access)

2. The space manager exhibits the problem when the B-tree is very wide and sparsely populated. The memory cache on the other hand only contains the available page extents (not the empty pages) and provides very fast access.

After placing the Exchange 5.5 Service Pack 2 on a machine exhibiting the initial space tree problem we saw much improved performance and 'receive queues' disappeared.

But, the improved performance only lasted for a short time; soon the queues start to build again but this time the distinctive high B+ Tree move nexts/prevs were not apparent.

The second space tree problem:

Exchange 5.5 Service Pack 2 seemed to work fine for a while; it was only applied to databases that had tables with wide space trees that contained few available empty page extents, this lead to a small 'in-memory' cache.

After a period of time, probably after extended periods of online defragmentation available extents typically of only one page are added to the 'in-memory' linked list causing the list to become large.

Compounding this problem is that the long value tables typically grow out to the right of a B-tree. The page numbers that are split are high, and each time a page is split a new page is requested. Because the space manager will first seek through the 'in-memory' linked list until a page number higher than the one being split is found, typically one will either be found close to the end of the linked-list or not found at all (in which case the first page in the list is returned).

In situations where the 'in-memory' cache is large the time taken to seek through the linked list can be prohibitively long. During this seek time the B-tree containing the page

being split is exclusively write locked preventing any other session (therefore user) making updates to it.

It is this behaviour, caused when a large space tree for a long value table has a high density of available page extents that the 'in-memory' cache becomes unable to satisfy the requests fast enough thus causing the symptoms displayed (Information Stores send/receive threads (4 by default) consume excessive CPU time (about 25% per thread))

The resolution to the Exchange 5.5 Service Pack 2 space tree problem

The design goal to provide locality when satisfying space requests has become prohibitively expensive as Exchange has scaled to larger and larger databases.

Analysis of performance data indicates the efficiency of the pre-read logic that makes use of locality provided by the space manager fails to make much gain of the ability to read multiple blocks in a single disk I/O. The gain provided by locality is far outweighed by the cost of expensive disk access and CPU time the space manager consumes in providing it.

Based on this evidence in Exchange 5.5 Service Pack 3 we have removed from the space manager the functionality to provide locality when issuing available pages.

We can understand how large the space cache for large tables becomes by monitoring the informational event that has been added. The 202 event is logged each time a space cache for a B-tree is built:

```
Event ID:    202
Source:      ESE97
Category:    Space Management
Description:

Database 'e:\exchsrvr\MDBDATA\PRIV.EDB': The database engine has built an
in-memory cache of 827 space tree nodes on a B-Tree (Objid: 24342,
PgnoRoot: 3015009) to optimize space requests for that B-Tree.  The space
cache was built in 156 milliseconds.
```

The Space Manager in Exchange 2000

The space cache will be eliminated in Exchange 2000 because ESE will be able to defragment space trees therefore solving the initial issue.

Buffer Management

One of the fundamental design goals of ESE is to avoid going disk, to do this ESE uses a comprehensive buffer manager.

The buffer manager has two principle jobs:

1. To decide how much memory the buffer cache should use, internally this feature is called Dynamic Buffer Allocation (DBA)
2. To decide which pages should remain in the buffer cache, ESE uses an algorithm called LRU-K to make this decision.

Dynamic Buffer Allocation

Introduced in Exchange 5.5, dynamic buffer allocation (DBA) has become a major factor in Information Store memory usage. Shortly after Exchange 5.5 shipped PSS had many calls asking about apparent memory leaks in the Information Store process (store.exe), in fact there were no memory leaks in store.exe ESE just changed the way it uses memory.

In 4.0/5.0 the Performance Wizard statically set the size of the cache. In 5.5 ESE observes the system and changes the size of the cache dynamically, the size of the cache can be observed using the ‘Cache Size’ counter on the ‘Database’ object in Performance monitor.

Each second ESE monitors the state of the cache, verifies the system’s requirements and makes adjustments to its size where necessary.

Design goals

The fundamental design goal is to maximize performance, we already know that going to disk is 1000s of times slower that getting the same data from memory, so the more effectively we can cache the better.

We want to maximize memory usage; Exchange 5.5 takes the view that it is much better to use almost all the memory on a server than leave memory unused.

Having said this we want to balance the needs of Exchange against the needs of the system as a whole, this means that whilst we try to use all the memory on the machine if other areas of the system need memory we will release it as fast as we can.

We only want to use the memory that we need, so we will not simply pre-allocate large areas of memory, instead we will consume memory as we need it.

General rules

So Dynamic Buffer Allocation uses 4 rules to govern the how large or small it should be:

- The more memory available the faster we will grow
- The more cache memory the faster we shrink
- The higher our memory load, the faster we grow
- And finally the higher the OS memory load the faster we shrink.

Exceptions

However DBA also considers the following exceptions to the rules above.

- Do not grow beyond the process working set size
- Do not grow if we have too many available buffers
- Do not shrink if the OS has too much available memory
- Fade in shrinking to release memory gradually

Formula

The formula DBA uses to calculate the size of the buffer cache is:

$$\frac{\partial C}{\partial t} = \frac{A}{T} \cdot \frac{\partial B}{\partial t} - \frac{C}{T} \cdot \frac{\partial P}{\partial t}$$

Where:

C = Cache Memory

A = Available Memory

T = Total Memory

B = Buffer Allocations

P = OS Page Allocations

TODO: Explain what this formula really means ;-)

The LRU-K replacement algorithm

Page Victimization / Page Replacement Policy

As we have seen DBA manages the size of the buffer and over time the buffer will become full with cached database pages; however there will be a need to cache more pages, to do this pages currently in memory will have to be removed from the cache to make room. Therefore we need a mechanism to determine which pages deserve to stay in the cache.

LRU

Traditionally database systems have used the LRU (Least Recently Used) algorithm, first described by Denning in 1968. When buffer space is needed LRU drops the page from the memory buffers that has not been accessed for the longest time.

However the LRU algorithm has a flaw, it decides what page to drop from memory based on too little information, limiting itself to only the time of last reference. Specifically LRU is unable to differentiate between pages that have relatively frequent references and pages that have very infrequent references.

LRU-K

The basic idea in LRU-K used since Exchange 5.5 is to keep track of the times of the last K references to memory pages (ESE sets the default value of K to be 2) and to use this statistical information to rank-order the pages as to their expected future behavior. Based on the statistical information the page replacement policy decision is made i.e. which memory-resident page to replace when a newly accessed page must be read into memory.

The LRU-K algorithm originally presented by O'Neil et al. in 1993 takes the idea of using the history of the last two references, or more generally the last K references, (where K is greater than or equal to 2) for each page when deciding which page should be victimized.

TODO: Explain by example

ESEUTIL

How fast is Eseutil

In Exchange 5.5 on 'reasonable' hardware we would expect speeds of around:

- 10 GB/hour for integrity (ESEUTIL /G)
- 3-5 GB/hour for repair (ESEUTIL /P)
- 4-5 GB/hour for defrag (ESEUTIL /D)

ESEUTIL in Exchange 2000

ESEUTIL in Exchange 2000 is still going through the development process. Exchange 2000 Beta 3 does not support the repair option and it is likely a graphical user interface to this utility will be built (probably based on the Microsoft Management Console).

The rest of this section contains the detail known at the time this document was written (Exchange 2000 Beta 3).

Changes to ESEUTIL in Exchange 2000

No more /IS, /ISPRIV, /ISPUB, /DS switches

The information about the location of databases and their log file paths is now stored in the Active Directory, not the registry so we can't easily extract the information we need. This means that the database, log file and system (checkpoint) paths have to be manually specified when defragging or recovering a database. If the log files are in a different directory, remember to use the /L parameter or you may get a -550 error. If the checkpoint is in a different directory use the /S parameter or you'll replay all logfiles.

Defrag (/D) does not shrink the streaming file

The streaming file is simply re-attached to the newly defragged database. NOTE: the size of the streaming file is reduced during online defragmentation.

No more /V /X switches with Integrity (/G) or Repair (/P)

Every time someone was running Integrity/Repair they were being told to specify these switches so they are now on all the time. Passing /V or /X into ESEUTIL will result in a usage error.

Verbose output goes to INTEG.RAW

When running ESEUTIL only the progress bars will be printed to the screen. All the verbose output will go to a file called INTEG.RAW. INTEG.RAW will be appended too by multiple runs of ESEUTIL.

Integrity/Repair support streaming files

When running /G or /P you must specify the streaming file with the /S option. In beta 2 we don't store the checksums of data in the streaming file so corruptions in the streaming file are not detected/repaired yet. The checksumming code is checked into RC1 though.

Integrity is multithreaded

The integrity check (and thus the first stage of repair) is multi-threaded. Each thread checks a different table in the database, the largest tables (Message and attachments) are checked first. Multi-threading combined with the new TTARRAY means that integrity check is over 4X faster in Platinum! This also means that the output of the integrity checks by the different threads is interleaved in INTEG.RAW. The output in INTEG.RAW will look like this:

```
Table1:     227:  checking table "Table1" (7)
Table2:     267:  checking table "Table2" (6)
Table1:     227:  checking data
Table2:     267:  checking data
Table2:     267:  checking index "Index1" (10)
Table2:     267:  checking index "Index2" (9)
Table1:     227:  checking index "Index1" (13)
Table2:     267:  checking index "Index3" (8)
Table1:     227:  checking index "Index2" (12)
Table2:     267:  rebuilding and comparing indexes
Table1:     227:  checking index "Index3" (11)
```

The multi-threaded output is of the form

```
table-name: thread-id: output
```

This makes it possible to see everything done on a particular table or by a particular thread. The output is harder to read though, and may require more post-processing.

Orphaned LVs are no longer ERRORs

There are now ERROR: and WARNING: messages. Orphaned LVs now show up as WARNING:'s so they won't show up if you search for ERROR:

Do not run Platinum repair against an Exchange 5.5 database

Doing so will change the database format and the database will not be usable in Exchange 5.5. You can run the integrity check (/G) but running /P will ruin the database.

Eseutil /mh priv1.edb and Eseutil /mh priv1.stm

The /mh switch can be used to dump the header of a .edb (Database) file, .stm (Streaming file) and .pat (Patch file)

Database (.edb) file

```
Initiating FILE DUMP mode...
      Database: PRIV1.EDB

         File Type: Database
   Format ulMagic: 0x89abcdef
   Engine ulMagic: 0x89abcdef
 Format ulVersion: 0x620,8
 Engine ulVersion: 0x620,8
     DB Signature: Create time:09/28/1999 12:15:27 Rand:3272303 Computer:
          cbDbPage: 4096
            dbtime: 97052 (0-97052)
             State: Consistent
     Log Required: 0-0
   Streaming File: Yes
         Shadowed: Yes
       Last Objid: 207
     Scrub Dbtime: 0 (0-0)
       Scrub Date: 00/00/1900 00:00:00
     Repair Count: 0
```

```
      Repair Date: 00/00/1900 00:00:00
  Last Consistent: (0x17,D,1DC)  11/23/1999 13:59:58
      Last Attach: (0x16,1F9C,1FF)  11/23/1999 13:57:05
      Last Detach: (0x17,D,1DC)  11/23/1999 13:59:58
             Dbid: 1
    Log Signature: Create time:09/28/1999 12:15:23 Rand:3246416 Computer:
       OS Version: (5.0.2128 SP 0)

Previous Full Backup:
        Log Gen: 22-22 (0x16-0x16)
           Mark: (0x16,1FC5,45)
           Mark: 11/23/1999 13:58:45

Current Incremental Backup:
        Log Gen: 0-0 (0x0-0x0)
           Mark: (0x0,0,0)
           Mark: 00/00/1900 00:00:00

Current Full Backup:
        Log Gen: 0-0 (0x0-0x0)
           Mark: (0x0,0,0)
           Mark: 00/00/1900 00:00:00

Current snapshot backup:
        Log Gen: 0-0 (0x0-0x0)
           Mark: (0x0,0,0)
           Mark: 00/00/1900 00:00:00

     cpgUpgrade55Format: 0
    cpgUpgradeFreePages: 0
cpgUpgradeSpaceMapPages: 0
```

Streaming (.stm) file

The field layout is the same as above, but the only fields that have valid data is:

```
Initiating FILE DUMP mode...
      Database: PRIV1.stm

         File Type: Streaming File
     DB Signature: Create time:09/28/1999 12:15:27 Rand:3272303 Computer:
         cbDbPage: 4096
            State: Illegal
         Shadowed: Yes
     Scrub Dbtime: 0 (0-0)
       Scrub Date: 00/00/1900 00:00:00
```

Database

Specifies the type of file being dumped, either Database, Streaming File or Patch file.

Database & Engine ulMagic

A hard coded value in ESE97 and ESE98 used during integrity check, should always be 0x89abcdef

Format & Engine ulVersion

Internal ESE version number to reflect version of ESE database format and ESE database access engine this database is working with. Higher version numbers should always recognize the database with lower version formats.

- ESE97 has a ulVersion of 620,2
- ESE98 in Exchange 2000 Beta 3 has a ulVersion of 620,8

DB Signature

A create time and random number used as a signature to match a database against a transaction log sequence. The database signature must match the value specified in the transaction log header for this database.

cbDbPage

The page size used in this database, Exchange used 4096 bytes, Active Directory uses 8192 bytes.

State

Either ‘Consistent’, the database has been shutdown cleanly or ‘Inconsistent’, the database is running (debug ese.dll only), has been shutdown un-cleanly or has just been restored from an online backup. If the database is in an ‘Inconsistent’ state there is data in the log files that is not yet in the database so the database file cannot be moved and the logs cannot be deleted. (See ‘Log truncation’ for details on which logs might be needed for recovery.)

Log Required

If a database is in an ‘Inconsistent’ State this field will contain the range of log files that must be present in the live transaction log directory for this database to be able to have run soft recovery successfully run against it. If the range of log files specified is not present recovery will not run and this database will not be mounted.

Streaming File

If the database file has an accompanying streaming file this field will be Yes. For the database to be successfully initialised the streaming file must be present and have the correct header.

Repair Count & Date

When & how many times repair (eseutil /p) has been run against this database since it was created (see the database signature create time). Running an offline defragmentation (eseutil /d) will reset this value.

Last Consistent

The date and time this database was last successfully stopped (Dismounted)

Last Attach

The date and time this database was last successfully mounted

Last Detach

TODO: How could this be different from Last Consistent

Dbid

A sequentially incremented number given to each database in the storage group

Log Signature

A create time and random number assigned to the transaction log sequence this database is working with. During recovery this signature must match the signature specified in the log file headers.

OS Version

The NT version at the time the database was last initialised. Because code page sort order can changed between NT versions all secondary indexes will be removed from the database when the NT version (inc. service pack) changes. It is not possible to run recovery (either hard or soft) on an inconsistent database that is on a version of NT different NT to the version specified in this field.

Eesutil /ml edb.log

```
Initiating FILE DUMP mode...
      Log file: edb.log
```

```
            lGeneration: 23 (0x17)
            Checkpoint: (0x17,15,1D9)
            creation time: 11/23/1999 13:59:39
            prev gen time: 11/23/1999 13:45:58
            Format LGVersion: (7.3703.1)
            Engine LGVersion: (7.3703.1)
            Signature: Create time:09/28/1999 12:15:23 Rand:3246416 Computer:
            Env SystemPath: F:\exchsrvr\mdbdata\
            Env LogFilePath: F:\exchsrvr\mdbdata\
            Env Log Sec size: 512
            Env (Session, Opentbl, VerPage, Cursors, LogBufs, LogFile, Buffers)
                (    272,   37800,    2320,   12600,      84,   10240,   65421)
            1 f:\exchsrvr\mdbdata\priv1.edb
              dbtime: 97021 (0-97021)
              objidLast: 206
              Signature: Create time:09/28/1999 12:15:27 Rand:3272303 Computer:
              MaxDbSize: 0 pages
              Last Attach: (0x16,1F9C,1FF)
              Last Consistent: (0x16,1F93,1B7)
            2 F:\exchsrvr\mdbdata\pub1.edb
              dbtime: 572673 (0-572673)
              objidLast: 5363
              Signature: Create time:09/28/1999 12:15:49 Rand:3308205 Computer:
              MaxDbSize: 0 pages
              Last Attach: (0x16,1FA1,1A1)
              Last Consistent: (0x16,1F9B,17A)
            Last Lgpos: (0x17,15,1EC)

       Integrity check passed for log file: edb.log

       Operation completed successfully in 1.688 seconds.
```

lGeneration

The log generation number, reported in decimal and hex. Once a file is rename from edb.log the hex based generation is used to create the file name, i.e. this case it would be edb00017.log

Checkpoint

TODO: The position of the checkpoint at the time the log file was created?

creation time

Date and time this log file was created

prev gen time

Date and time the previous log file in the generation sequence was created

Signature

A create time (of the log sequence i.e. edb00001.log) and random number used as a signature to match a transaction log sequence against the databases it is working with.

Env SystemPath

Location the storage groups system file (i.e. edb.chk and temporary .edb file)

Env LogFilePath

Location of the live transaction log files

For Each database in the storage group that was attached when the log file was created:

Hard coded path to database file

In the above example f:\exchsrvr\mdbdata\priv1.edb, it is important to note that if the database priv1.edb is inconsistent it cannot be moved from the f:\exchsrvr\mdbdata directory.

dbtime

The dbtime of the database at the time of log file creation

Signature

The Database signature of the database at the time the log file was created

Last Attach

The log generation and offset the last time the database was mounted (at the time the log file was created)

Last Consistent

The log generation and offset the last time the database was successfully dismounted/stop (at the time the log file was created)

Last Lgpos

TODO: What is the difference between this and the first field in the dump 'Checkpoint'

Eseutil /mk edb.chk

```
Initiating FILE DUMP mode...
      Checkpoint file: edb.chk

      LastFullBackupCheckpoint: (0x0,0,0)
      Checkpoint: (0x17,15,1D9)
      FullBackup: (0x17,8,16)
      FullBackup time: 11/23/1999 13:59:37
      IncBackup: (0x14,8,16)
      IncBackup time: 11/23/1999 13:44:40
      Signature: Create time:09/28/1999 12:15:23 Rand:3246416 Computer:
      Env (Session, Opentbl, VerPage, Cursors, LogBufs, LogFile, Buffers)
          (    272,   37800,    2320,   12600,      84,   10240,   65421)

Operation completed successfully in 0.140 seconds.
```

LastFullBackupCheckpoint

TODO: Is this used

Checkpoint

The first hex value (0x17 in this case) is the transaction log generation the checkpoint is currently pointing at. All data for currently mounted databases in log file generations before this number have been successfully committed into the database files. Please see the section for 'Log truncation for more detail'. The 2 other values (15, 1D9) are internal offsets

FullBackup

The value of the checkpoint at the time the last full or copy backup started

FullBackup time

The time the last full or copy backup started

IncBackup

The value of the checkpoint at the time the last incremental or differential backup started

IncBackup time

The time the last full incremental or differential backup started

Signature

The signature of the transaction log sequence this checkpoint points into.

ESEFILE

ESEFILE is a utility written by the ESE development team although it contains no JET code. ESEFILE has a range of miscellaneous uses; here each use is described.

Large and Fast File copy

ESEFILE /C can copy a file of any size, copying a file this way is very fast. For large (>100MB) files it can be twice as fast as COPY/XCOPY.

Usage

```
ESEFILE /C source destination
```

The file is opened un-cached so we can copy files larger than can be opened in cached mode. There are several caveats:

- Wildcards are not accepted
- Only one file can be copied at a time
- The destination file cannot be on a networked drive
- File timestamps are not preserved

Checksumming an entire database

To checksum all the pages in a database whilst it is offline you can use ESEFILE.

Usage

```
ESEFILE /S database (ESE format -- Exchange 5.5)
ESEFILE /X database (EDB format -- Exchange 4.0/5.0)
```

This is not a substitute for the integrity check! Logical verification is not done by ESEFILE. If you use /X on an ESE format database or /S on an EDB format database you will get a checksum error on every page.

Checksumming an individual page

To verify the checksum of an individual page use ESEFILE /d

Usage

```
ESEFILE /D database pageno
```

Output

```
00000000  2a c3 aa 89 01 00 00 00 01 00 00 00 00 00 00 00 *...............

00000010  00 00 00 00 00 00 00 00 01 00 00 00 c4 0f 00 00 ................
… The binary value of the page …

checksum: 0x5477CEA7 (1417137831) (INCORRECT)
```

Large File deletion

To delete a file larger than can be deleted using explorer or any of the command line utilities provided with Windows use ESEFILE /u

Usage

```
ESEFILE /U filename
```

ESE Parameters

This is the complete list of configurable parameters in ESE98. Since ESE98 no longer uses the registry, the value of these parameters must be set by the ESE client application (i.e. the Information Store).

Each ESE client will have different methods of setting these parameters, the information store sets many of these values based on parameters calculated during the running of the Performance Optimiser Wizard

JET_paramSystemPath

Path to the checkpoint file directory. The default location for the checkpoint file is the current directory.

JET_paramTempPath

Directory where temporary files may be created. The default location for temporary files is the current directory.

JET_paramMaxSessions

Specifies the maximum number of user sessions that can be simultaneously open.

JET_paramMaxOpenTables

Specifies the maximum number of tables which can be simultaneously open.

JET_paramMaxVerPages

Specifies the maximum number of modified pages allowed before JetCommitTransaction or JetRollback must be called, on a per-instance basis.

JET_paramGlobalMinVerPages

Specifies the minimum number of pages reserved for modified pages, globally, not per-instance based. The version store may grow beyond this value, limited by available memory and the per-instance JET_paramMaxVerPages.

JET_paramMaxCursors

Specifies the maximum number of cursors which can be simultaneously open.

JET_paramLogFilePath

The path to the directory where log files are stored.

JET_paramLogBuffers

Specifies the number of log buffers.

JET_paramLogFlushThreshold

Specifies the number of log buffers dirty before the log buffers will be flushed. (Not supported)

JET_paramCacheSizeMax

Specifies the maximum number of bytes to use for database page buffers. Default value is 512.

JET_paramCacheSizeMin

Specifies the minimum number of bytes to use for database page buffers. Default value is 64. When it is the same as JET_paramCacheSizeMax, the DBA is turned off.

JET_paramStartFlushThreshold

Evictable pages at which to start a flush. Default value is either 100 or proportional to CacheSizeMax.

JET_paramStopFlushThreshold

Evictable pages at which to stop a flush. Default value is 400 or proportional to CacheSizeMax.

JET_paramWaitLogFlush

Specifies the number of milliseconds of wait time prior to flushing the log, on commit transaction to transaction level 0. This wait allows other users to who commit transaction to transaction level 0 during the wait, to share the log flush with the other users. (not used in ESE98)

JET_paramExceptionAction

Specifies whether internal exceptions should be caught (causing a dialog box to pop up) or passed on to the calling application. Non-zero means that exceptions should be caught, while zero means that exceptions will not be caught. The default is not to catch exceptions.

JET_paramEventLogCache

Specifies the size of event log cache. If the event log cache is available, ESE will defer the event logs into the cache and log the cached events when event log service becomes available next time when it tries to log an event. When the cache is full, it will start to throw away the event logs. By default, the parameter is set to 0, so it will return error if event service is not available.

JET_paramEventSourceKey

Points to the application's event source key. By default, the event source key is ESE98. To set this parameter, user must have the event source registry key properly set.

JET_paramEventSource

Points to a language independent stream. This stream will be logged in to the event log records. Typically this stream is set to application name such as "Exchange IS".

To do

Bibliography and References

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD Conference 1995: 1-10

Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum: The LRU-K Page Replacement Algorithm For Database Disk Buffering. SIGMOD Conference 1993: 297-306

Salmon, Wesley C. ed. Zeno's Paradoxes. New York: The Bobbs-Merrill Company, Inc., 1970.

J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, 1993.

Adelson-Velskii, G. M., and E. M. Landis. "An Algorithm for the Organization of Information." *Soviet Math. Doclady 3*, 1962, pp. 1259-1263.

Laurion Burchall: 1-403: Exchange 5.5 Database technology (MEC 99)

Laurion Burchall & Philip Hupf: 1-402: Exchange 2000 Database technology (MEC 99)

Laurion Burchall: Logfile replay precautions in Exchange Server 2000: Thu 11/11/99 8:30 PM

Laurion Burchall: RE: JET_errVersionStoreOutOfMemory question: Wed 9/8/99 1:21 AM

Laurion Burchall: RE: Exchange and Snapshot Isolation: Thu 6/17/99 5:34 PM

Laurion Burchall: Logfiles in Platinum Beta 2: Thu 5/20/99 7:52 PM

Laurion Burchall: ESEUTIL in Platinum Beta 2

Laurion Burchall: EDBTMP.LOG: Wed 8/12/98 7:54 PM

Laurion Burchall: ESEUTIL speed: Tue 4/28/98 11:17 PM

Laurion Burchall: ESEUTIL /MS: Fri 4/24/98 6:59 AM

Laurion Burchall: Long-Values and orphaned LVs: Mon 4/6/98 8:57 PM

Jonathan Liem & Laurion Burchall: RE: "Best Practices Whitepaper" -- question on .pat files and restoring from Online backup

Jonathan Liem: RE: Event 202: Thu 10/28/99 12:53 AM

Steven Gott: RE: Storage question (new thread): 01 November 1999 17:27

Perry Clarke: 1-304: 'Exchange 2000 performance and scalability'

Misc – Notes to myself

TODO: What do we use the patch file for during a snap

TODO: Why do we continue to allow write to the stream file during Start and Stop

TODO: Remember we force a log generation when we start RestoreComplete

I haven't described single instance storage

X5: 122259, we can move databases without invalidating backup

131534: ESE98: utility to indentify the location of a -1018 on the physical nt volume

Added code to eseutil to allow dumping of the Restore.Env (eseutil /cm ...)

Added code to call HrESERestoreComplete() for an already restored backup (based on the generated Restore.Env) - eseutil /cc ....

Don't stop the backup on errors on one database, allow skipping the database
We mark the database with error not part of the backup and continue the backup (previouse we stopped the backup session at JET level)

I haven’t described exactly what temp.edb is used for (how will we cope with multiple databases from potentially different storage groups in the same directory, will temp.edb change its filename/will it clash?)

Need to explain records in a clear manner

Do really have to restore the full backup set last?

Small packet communication: Eproxy Shared memory IO completition port interface - targeted at solving the small packet communication issue.

Inbound and outbound queue per protocol, as little as 1 context switch for 5 round trips...

Good set of perfmon counter for diagnosing bottlenecks

Large packet communication:

IOCP???

Doesn't use EPOXY.

Use files, IIS to write to file and then just pass the handle over to Store, commited to disk and EPOXY passes over

the handle and where the data is... and logged, response is then passed back to EPOXY

Extended Attribute name (EA name)

SMTP -> to disk -> file handle cache -> EA name

Client -> IIS -> Store --- < Store returns EA name

Fast drive can do about 100 4k IOs randoms a second (500/1000 sequential IOs per second)

Stm file does sequential IOs, better than JET at storing large blocks

The IIS protocols buffer up to large size... (at about 64k difference betwene ran and seq become less important)

Prevents fragmentation of messages on disk.

OLD for STM (creates free empty blocks of 2mb chunks so ESE can issue large chunks)

STM online defrag is an inversion of classic disk defrag algorithms, committed pages are moved towards the end of the file and converts the file into a circular queue.

Keeps IOs down to one per message.

Control OLDSLV

'Zero out deleted database pages', will zero out slv as well which will get compress away on backup

File handle cache... 1. it is cross protocol, no need for createfile... just go to store and get EA. (EA is used as cache)

There is a name cache (protocol specific)... which maps to the EA

IFS takes advantage of NT so we can cache data.

Underlying SLV file is opened unbuffered... all the caching is done a per file/message basis

3000 views when dealing with small files/messages.. limitation.

When a file handle is open, the memory manager will keep pages faulted out of the cache on the standby list...

Cache manager views is filled up, it needs to be evicted... is gets removed from Cache managers bytes, but it stays on the stand by list, still in memory.

Reading a message in standby list results in a cache fault (and page fault) but no page read as memory manager executes a transition fault to return pages to cache manager’s working set

Standby list managed by memory manager ... is an LRU of the pages recently used, is part of available bytes... (a component of available bytes)

Using soft faulting... saves having to go to disk. (allows us to get round the address space problem), Stand by list is not limited by address space limits.

Standby list is counted by available bytes... at the same time it is still important for ESE and Jet to cache the meta data, which involves multiple reads.. .where are stm stuff is just one read.

So ESE will squeeze the standby list.

However we can take advantage of memory > 4Gb cache manager is not taken part 4Gb limit of the address space nor is the stand by list

Win32 - Don't open and close files... breaks our scheme for keeping files contgous, buffer up the writes between the same CreateFile/Close

Rapid rates of creates and deletes will cause the STM to fragment

Setting FileFlagTemporary and DeleteOnClose flags in Create file is very cheap, cheaper than the temp flags in NT... there is no need for round trips to the store

VA.... DBA did not pay attention to limits of Address space.

DBA 2 will maintain 64mb of available address space... there will always be <<. Calls will never fails becasue of AS issues

Version store used loads of AS... VS had to be able to get AS so we allocated a lot up front, but because of changes in DBA we can now do this dynamically.

Stack size down to 128Kb

SLV data is not buffered in store process... it is done by cache manager and stand by list.

We have freed upto 1Gb of addr space.

IFS - Power failure... IFS pages are freed on startup... not until the B Trees in JET are updated.

Perf goals

MAPI services - comparable per server to 5.5... upgrade in place

Internet protocols -> factors 2 or 3

Beat MCIS or Exchange 5.5 which ever is higher.

ESP Tool - Exchange Scalability and Performance

3% change using 8k page size, not worth changing it (using 32k message size)

Hardware support for ESE

Explicitly state what platforms ESE is not supported on i.e. software redirectors etc.

SQLHDTST

Place this on page 19

What is Escrow

Exchange 2000 ESEUTIL retail options; Exchange 2000 ESEUTIL debug options

Exchange 2000 ESE Perfmon counters